БИБЛИОТЕЧКА ПРОГРАММИСТА

В. М. ПЕНТКОВСКИЙ

Язык программирования Эль-76

БИБЛИОТЕЧКА
ПРОГРАММИСТА

В. М. ПЕНТКОВСКИЙ

ЯЗЫК ПРОГРАММИРОВАНИЯ Эль-76

ПРИНЦИПЫ ПОСТРОЕНИЯ ЯЗЫКА И РУКОВОДСТВО К ПОЛЬЗОВАНИЮ

ИЗДАНИЕ ВТОРОЕ,
ИСПРАВЛЕННОЕ И ДОПОЛНЕННОЕ

МОСКВА «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1989

ББК 22.18
П25
УДК 519.68

Серия основана в 1968 году
Выпуск 61

Пентковский В. М. **Язык программирования Эль-76.** Принципы построения языка и руководство к пользованию.— 2-е изд. испр. и доп.— М.: Наука. Гл. ред. физ.-мат. лит., 1989 (Б-чка программиста).— ISBN 5-02-013982-3.

Знакомит с языковыми основами построения многопроцессорной вычислительной системы Эльбрус, концепциями создания базового языка программирования высокого уровня Эль-76 и его ролью в использовании системы, а также с принципами разработки базовых средств поддержки современных систем программирования на языках высокого уровня. Дается полное описание Эль-76, а также определяются технические детали, необходимые для всех уровней практического программирования на этом языке.

Для программистов и читателей, интересующихся современными тенденциями в развитии вычислительной техники.

Табл. 2. Ил. 16. Библиогр. 107 назв.

П $\frac{1404000000-106}{053\,(02)-89}$ 145-89

ISBN 5-02-013982-3

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

В предлагаемой книге изложены принципы построения языка программирования Эль-76, дано его полное описание, а также приведен ряд дополнительных материалов, необходимых в практической работе с языком. Таким образом, книга в основном посвящена вопросам программирования в системе Эльбрус и вопросам развития алгоритмических языков. Можно отметить, что подобная специализация изложения не позволяет исчерпывающим образом охарактеризовать особенности данной системы. Поэтому выходу книги должно было бы предшествовать издание, достаточно полно и всесторонне отражающее тематику Эльбруса в целом. Чтобы в какой-то степени заполнить этот пробел, в предисловии и в начале первой главы приведены некоторые сведения наиболее общего характера, позволяющие читателю получить представление о той среде, в которой разрабатывался язык.

Вычислительная система Эльбрус разработана в Институте точной механики и вычислительной техники АН СССР им. С. А. Лебедева (ИТМиВТ) коллективом инженеров, конструкторов, создателей САПР и системных программистов. Разработка системы носила интегральный характер, т. е. одновременно разрабатывались архитектура и структура машины, структура операционной системы, структуры системы программирования и язык общения с системой — Эль-76, являющийся языком высокого уровня. Как видим, под термином «вычислительная система» подразумевается единое по замыслу образование, состоящее из ЭВМ — высокопроизводительного *многопроцессорного вычислительного комплекса* (МВК) Эльбрус — и *общего системного программного обеспечения* (ОСПО) этого комплекса.

ОСПО состоит из двух основных компонентов: операционной системы и системы программирования. Первая обеспечивает поддержку многопроцессорности, многозадачный и многопользовательский режим разделения времени, телеобработку и пакетную обработку, управление параллельными процессами, памятью и универсальным архивом. В состав системы программирования входят трансляторы языка Эль-76 и распространенных языков высокого уровня, а также единые инструментальные компоненты (система аварийной выдачи и отладки, редактор, комплексатор программ и пр.). Система

программирования является многоязыковой в том смысле, что имеет единые инструментальные компоненты, обеспечивает возможность вызова программы, написанной на одном языке, из программы, написанной на другом языке, и построена на основе единых базовых средств поддержки. Новые возможности ОСПО включают средства управления базами данных, средства управления конфигурациями и версиями программных проектов, а также полиэкранный и графический режимы работы пользователя за терминалом. Операционная система, транслятор Эль-76, инструментальные компоненты многоязыковой системы программирования, а также базовые средства поддержки систем программирования разработаны в ИТМиВТ; разработка ОСПО в части трансляторов Алгола, Фортрана и других основных языков программирования проводилась в Новосибирском филиале (НФ) ИТМиВТ. Интерпретирующие и компилирующие системы для языков символьной обработки и языка Паскаль разрабатывались в Ленинградском государственном университете (ЛГУ), компилятор языка Симула-67 — в Ростовском государственном университете (РГУ).

Положительный опыт предшествующих проектов, в том числе опыт ведущих отечественных разработок в области вычислительных машин, операционных систем и систем программирования, обеспечил для МВК и ОСПО Эльбрус хорошую базу, которую можно было рассматривать как основу для дальнейшего развития. В этой связи надо прежде всего упомянуть серию работ по созданию высокопроизводительных универсальных ЭВМ, приведших к построению ЭВМ БЭСМ-6, а также принцип структурной интерпретации языков высокого уровня, выдвинутый В. М. Глушковым и нашедший свое начальное воплощение в ЭВМ серии МИР и проекте машины «Украина» [67].

В системе Эльбрус решение проблем программного обеспечения существенно связано с решением задач инженерного характера, и провести границу между ними сложно. Однако, в связи с конкретностью назначения книги, система (и, в частности, аппаратура) рассматривается в ней с точки зрения вопросов программирования. Инженерные решения пояснить в рамках данной работы, к сожалению, не представляется возможным. Основные теоретические и практические положения отражены в публикациях ведущих разработчиков этих разделов системы Эльбрус [2, 4—6].

В первой главе книги вводятся концептуальные основы данной работы. Анализируются базовые механизмы языков высокого уровня (ЯВУ), вводятся элементы гипотетического ЯВУ, отличительными особенностями которого является сочетание динамического управления типами с механизмом абстрактных типов данных. На разработку элементов гипотетического языка оказали влияние идеи ортогональности построения языков Лисп и Алгол-68, работа

Ф. Бауэра по основам информатики [55], концепция эквивалентности типов по наименованию, введенная в языке Паскаль. В главе также определяются требования к ЯВУ, предназначенному для программирования систем. С позиций разработки систем программирования на ЯВУ вводятся модели реализации механизмов гипотетического языка; эти модели называются «структурными моделями» программы и памяти выполняемой программы. На этих моделях показано не только выполнение основных конструкций языка, но также функционирование системы программирования (СП), общающейся с пользователем в терминах исходной программы, что особенно важно в ходе отладки и в случаях, когда необходимо произвести аварийную выдачу сообщения об ошибке и о текущем состоянии процесса выполнения программы. Далее в главе анализируется соотношение между архитектурой ЭВМ, с одной стороны, и разработкой ЯВУ и систем программирования, с другой; обсуждаются ограничения распространенных ЯВУ по сравнению с механизмами гипотетического ЯВУ и причины этих ограничений; анализируются причины осложнений, возникающих при разработке СП, общающихся с пользователем в терминах исходной программы, причины ненадежности ЯВУ и их реализации. Отмечено, что во всех случаях основной причиной является семантический разрыв между механизмами ЯВУ и архитектурой традиционных ЭВМ. На основе анализа тенденций развития архитектур отмечено, что одно из перспективных направлений связано с устранением семантического разрыва, а также то, что целесообразна совместная разработка нового языка и новой архитектуры, ориентированной на базовые механизмы языков, а не на конкретный существующий ЯВУ.

Вторая и третья главы содержат описание языка Эль-76, который можно рассматривать как конкретизацию гипотетического языка, изложенного в гл. 1. Во второй главе вводится ядро Эль-76: средства конструирования типов, описания, операторы и выражения. Третья глава целиком посвящена вопросам взаимодействия программы с внешними объектами (с файлами, с архивом, с другими программами). Четвертая и пятая главы, а также приложения содержат справочно-информационный материал, который может понадобиться в практической работе с языком (атрибуты объектов, средства обработки внутреннего представления объектов, синтаксис языка и пр.).

Автор не ставил перед собой цель разработать учебник языка. Однако предпринят ряд мер, которые, возможно, облегчат изучение языка. Для этого сделано следующее:

1) основное описание разбито на две части (гл. 2, 3), причем во вторую часть (гл. 3) вынесено описание таких средств языка, которые могут понадобиться в основном при программировании алгоритмов, активно взаимодействующих с внешними объектами;

2) материал каждой главы разбит на основной и второстепенный. Второстепенные сведения отделены от основного текста знаками ■.

На первом этапе знакомства с языком будет достаточно основных сведений первой части (§ 1—13 гл.2). Сюда, в частности, входят простейшие способы форматного ввода/вывода и элементарные приемы составления пакетных заданий. Примеры, приведенные в конце этой части (§ 20 гл.2), также базируются на материале основного текста. Иными словами, разделы и примеры, входящие в основной текст гл. 2, подобраны так, чтобы программист, знакомый с программированием на алголоподобных языках, мог, не изучая всего описания, провести аналогию между Эль-76 и знакомым ему языком программирования. В силу универсальности языка этих же сведений достаточно, чтобы написать не только простую программу, но и пакетное задание для ее запуска. При необходимости программа может содержать операторы обмена, а пакетное задание — определение входных и выходных файлов.

Отметим, что, несмотря на подобное разграничение, описание можно читать подряд.

По-видимому, имеет смысл сказать несколько слов об основных этапах развития языка, поскольку его история отражает эволюцию системы в целом и объясняет отличие второго издания книги от первого. Предварительный вариант системы команд и языка появился в 1972—1973 гг. После выпуска описания первой версии Эль-76 в 1973 г. были предприняты первые эксперименты по использованию языка для программирования операционной системы и трансляторов алгоритмических языков. На основании полученного опыта программирования и опыта реализации языка было признано целесообразным внести некоторые изменения в систему команд и в Эль-76. В частности, стало очевидным, что есть возможность дополнительно повысить уровень языка, не снижая при этом эффективность рабочих программ. В связи с этим в 1976—1977 гг. была введена новая версия системы команд и выпущено описание новой версии Эль-76. Этот этап развития системы особенно отчетливо свидетельствует о взаимовлиянии вопросов разработки аппаратуры, программного обеспечения и языка.

Начиная с 1977 г. ядро языка в достаточной степени стабилизировалось, но в процессе развития и эксплуатации системы возможности языка регулярно расширялись. Последним значительным этапом было введение средств конструирования новых типов. В первом издании книги было описано ядро языка, базирующееся на полностью динамическом управлении типами, причем номенклатура типов была ограничена стандартными аппаратно-распознаваемыми типами. С точки зрения разработки языка этот этап являлся наиболее важным, и практика применения языка с динамическим управлением и контролем типов для построения больших программных

систем, в том числе — операционной системы, должна была показать, правильно ли выбраны исходные позиции при создании Эль-76.

Полученный опыт эксплуатации языка подтвердил удобство использования подобного языка в области профессионального программирования больших программных комплексов. После этого язык был расширен механизмом конструирования новых типов и средствами управления степенью динамизма в части контроля типов. Это позволяет программисту выбирать наиболее адекватный для данной программы или фрагмента программы способ управления типами: от полностью статического до полностью динамического контроля типов. Расширенный язык и принципы его построения описаны во втором издании книги. Гл. 3 и 4 были приведены в соответствие с текущим состоянием реализации системы файлов.

Объем книги не позволил включить средства «определяемого синтаксиса», с помощью которых разработчик программного комплекса (в частности, пакета прикладных программ) может вводить собственный входной язык управления этим комплексом. Принципы разработки этого раздела Эль-76 описаны в [24], а примеры использования этих средств приведены в книге В. О. Сафонова о методах программирования в системе Эльбрус, которая выходит одновременно с данным изданием.

* * *

В предлагаемом языке и книге отражены результаты труда большого коллектива инженеров и программистов. Поэтому автор считает необходимым назвать по крайней мере тех из них, кто оказывал наибольшее влияние на формирование и практическое внедрение языка, а также тех, кто помогал ему при подготовке публикации.

В первую очередь надо подчеркнуть, что за разработкой языка стоит разработка тех функциональных возможностей машины и операционной системы, которые связаны с развитием и реализацией фундаментальных понятий алгоритмических языков и процессов обработки информации. Идеи, составившие основу этой части проекта Эльбрус, и в том числе положения, обусловившие основные черты Эль-76, были предложены руководителем разработки архитектуры машины и программного обеспечения Б. А. Бабаяном. Ведущими разработчиками компонентов ОСПО, определивших возможности языка, являются: С. В. Семенихин, С. В. Веретенников, В. Ю. Волконский, С. М. Зотов, А. И. Иванов, Ю. С. Румянцев, В. П. Торчигин, М. И. Харитонов, В. С. Шевяков; энергичную поддержку работ и координацию обеспечил А. Л. Плоткин.

Эта книга не является учебником по Эль-76. Она написана автором языка, руководившим разработкой созданных в ИТМ и ВТ разделов многоязыковой СП. Поэтому выделены проблемы проектирования Эль-76, СП и описания языка. Разработка опреде-

ляемых типов Эль-76 выполнена с Б. П. Синдеевым и руководителем разработки транслятора Эль-76 Ю. С. Румянцевым. Разработка структурных моделей программы и памяти, положенных в основу построения базовых средств технологической поддержки системы программирования, выполнена с В. Ю. Волконским, который руководил созданием компонентов ОСПО, реализующих эти средства.

Разработка языка происходила в тесном контакте с инженерами-разработчиками МВК Эльбрус, с создателями трансляторов Эль-76 и с пользователями языка. Этими нервыми и весьма требовательными пользователями, написавшими и отладившими около 500 тысяч строк достаточно сложных программ на Эль-76, были разработчики ОСПО, в том числе квалифицированный коллектив системных программистов НФ ИТМ и ВТ, руководимый Г. Д. Чининым. Возможность непосредственного взаимодействия со всеми создателями системы, высказанные ими предложения и критика недостатков крайне помогли в работе над языком и его описанием. Всем им автор приносит свою искреннюю благодарность, а также тем, кто способствовал появлению этой книги. Прежде всего Н. Б. Малышеву, Н. В. Мартыновой и В. П. Синдееву за подготовку примеров § 20 гл. 2, исправлений гл. 3, 4 и ряда приложений; Т. П. Новиковой, оказавшей неоценимую помощь при подготовке рукописи первого издания, и помогавшей ей С. К. Борисовой.

Автор глубоко признателен редактору первого издания книги — одному из основателей и руководителей НФ ИТМ и ВТ А. П. Ершову, а также С. С. Лаврову за плодотворные обсуждения концептуальных аспектов разработки, позволившие автору глубже раскрыть принципы построения языка, за критические замечания по первоначальному варианту рукописи первого издания и материалам второго издания, которые послужили основой для их переработки, а также за постоянное внимание и активную поддержку работ, связанных с программным обеспечением системы Эльбрус.

Особенно я обязан Б. А. Бабаяну. Именно в ходе многочисленных бесед с Борисом Арташесовичем формировалось мое представление о назначении и средствах языка, а также складывался и отрабатывался материал разделов книги, посвященных принципам его построения.

ГЛАВА 1

БАЗОВЫЕ МЕХАНИЗМЫ ЯЗЫКОВ ВЫСОКОГО УРОВНЯ И ВЛИЯНИЕ АРХИТЕКТУРЫ ЭВМ НА РАЗРАБОТКУ ЯЗЫКОВ

1. Введение

Сегодняшние требования к языкам программирования систем определяются нуждами разработки больших программных комплексов (БПК) повышенной сложности. Роль БПК в научных исследованиях и промышленности постоянно растет. К числу основных областей относятся: управление в реальном масштабе времени индустриальными процессами и экспериментом, обработка результатов эксперимента, моделирование и автоматизация проектирования ЭВМ, летательных аппаратов и других устройств, телекоммуникации, электронное обслуживание.

Целесообразность применения языков высокого уровня (ЯВУ) для программирования сложных систем обусловлена, как известно, следующими обстоятельствами. Для БПК чрезвычайно важны показатели надежности, удобства отладки и сопровождения программ, а также их модернизации в ответ на изменяющиеся внешние требования и обстановку эксплуатации. В качестве одного из основных инструментов для удовлетворения этих нужд уже сегодня можно рассматривать процедурные ЯВУ. Наиболее ценными свойствами этих языков являются: ориентация структур управления и данных на описание алгоритма, а не на специфику аппаратных возможностей традиционных ЭВМ, контроль правильности обработки данных в соответствии с их типами и структурой, средства разработки модульных программ.

По сравнению с традиционным ассемблерным программированием, где отсутствуют основные элементы типизации данных, контроля их обработки и защиты данных и программ, эти свойства ЯВУ дают упрощение программирования не только в процессе написания программ, но, что не менее важно, облегчают их отладку и модерни-

зацию. По некоторым оценкам [9] применительно к сложным программам ЯВУ обеспечивает повышение производительности программиста в несколько раз (в статье приводится коэффициент 8÷10, и тут же отмечается, что полученная в результате компиляции машинная программа обычно в 2÷5 раз длиннее по сравнению с той же программой, написанной на языке Ассемблера искусным программистом; кроме того, первая обычно расходует больше памяти и работает медленней, чем соответствующая программа, составленная вручную).

В понятие программирования на ЯВУ включается не только написание программы, но также весь цикл работы с ней*), включая автономную и комплексную отладку, выявление и исправление ошибок в процессе сопровождения и т. д. Подразумевается, что на всех этапах программист работает с программой исключительно в терминах используемого ЯВУ. Можно сказать, что лишь при таком подходе, основанном на применении адекватных средств поддержки разработки сложных систем на ЯВУ, можно полностью извлечь потенциальную выгоду, заложенную в ЯВУ. В противном случае неполная поддержка всего цикла программирования может привести к отрицательному эффекту.

Если, например, исходный текст пишется на ЯВУ, а отладка производится в машинных кодах или в терминах Ассемблера, то это означает, что от программиста требуется знание особенностей машины и умение произвести обратное (по отношению к тому, что выполнил компилятор) отображение кодов в языковый текст, чтобы внести исправление в исходную программу. В ряде случаев из-за того, что цикл внесения исправлений усложнен, могут непосредственно правиться машинные коды. Если при этом изменения не отслежены в исходном тексте, то ближайшая перекомпиляция программы приводит к потере правки кода, а, следовательно, к тому, что обнаруженная и один раз исправленная ошибка по-прежнему сохранится в программе.

Проблемы применения языков высокого уровня. Несмотря на известные преимущества ЯВУ по сравнению с Ассемблером, на практике оказывается, что в ряде наиболее трудоемких приложений эти языки практически не применяются или применяются частично.

---

*) В [53] приводится следующее распределение затрат на цикл разработки программного обеспечения (за исключением этапа сопровождения): 1/3 — анализ требований, разработка спецификаций, проектирование, 2/3 — написание и отладка, т. е. этапы, на которых язык программирования играет принципиальную роль. Сопровождение может потребовать в два раза больших затрат, чем цикл разработки, но и здесь роль языка не менее важна, так как наряду с управлением версиями сопровождение включает обнаружение и исправление наиболее сложных ошибок, проявляющихся в реальных условиях эксплуатации, а также модернизацию системы.

Можно считать, что основная причина этого явления состоит в том, что традиционные ЭВМ, архитектура которых происходит от архитектуры Неймана, недостаточно хорошо приспособлены для программирования сложных систем на ЯВУ. Это в свою очередь проистекает из-за так называемого «семантического разрыва» между архитектурой ЭВМ и основными механизмами ЯВУ [54, 80].

Несмотря на то, что процедурные ЯВУ и традиционные ЭВМ следуют общему принципу операционной обработки информации [55], есть ряд существенных отличий:

— единое для программ и данных неструктурированное (или линейное) общедоступное адресное пространство в отличие от принятых в ЯВУ механизмов разделения программ и данных*), разделения объектов на доступные и недоступные в данной точке программы (контекстная защита в ЯВУ), средств построения структурированных объектов;

— хранение в памяти данных в виде последовательности битов без аппаратно-распознаваемой информации о смысле этих данных и, соответственно, обработка данных без учета смысла, т. е. бесконтрольная обработка, в отличие от принятого в ЯВУ принципа типизации объектов и контроля правильности их обработки в соответствии с типом (из которого, в частности, следует различимость программ и данных).

Для пользователя ЭВМ семантический разрыв выражается в том, что существует ряд проблем применения ЯВУ. Перечислим основные из них:

1. Потери эффективности при переходе от языка Ассемблера к ЯВУ, а именно увеличение объема кода и замедление программы. Основная причина состоит в том, что для компенсации семантического разрыва базовые конструкции ЯВУ кодируются последовательностью нескольких команд и возрастает объем памяти, требуемый для хранения данных по сравнению с тем, как это мог бы устроить программист, знающий особенности машины.

Например [54], оптимизирующий компилятор языка ПЛ/1 (IBM) кодирует присваивание $c(к,м)=a(к,м)+в(м,к)$ в 17 команд (общим объемом 62 байта), если программист задал отключение контроля границ. В противном случае длина кода составляет 75 команд (274 байта), причем 40 команд тратится на контроль границ. Первый вариант можно приравнивать к случаю ассемблерной программы, а второй характерен для ЯВУ. Объем кода увеличился на 340%. При этом возросло число битов информации, передаваемых между процессором и памятью. Как следствие, оказывается, что введение контроля замедляет программу в 3 раза.

*) Даже в языке Лисп, где программу можно обрабатывать как данные, такой объект имеет особый признак, отличающий его от объектов, не являющихся программами.

Похожие результаты приводится Уичменом [56, 57]:

— интерпретация таких языковых черт, как контроль границ массивов и контроль арифметического переполнения для языка Алгол-68 на ЭВМ ICL 1900, снижает производительность машины (для программиста) в два раза;

— сравнение реализаций рекурсивного вызова процедур показывает значительную нестабильность по числу команд в зависимости от архитектуры ЭВМ, языка и компилятора. Реализация этой конструкции может оказаться в три и более (до 10—100) раз менее эффективной, чем вызов подпрограмм на языке Ассемблера.

2. Следующая трудность, проявляющаяся при программировании БПК, это разного рода ограничения средств управления объектами и типами, присущие всем распространенным ЯВУ (подробно см. § 13). Эти ограничения вносились в языки с целью достичь удовлетворительной эффективности, по крайней мере для определенного класса задач. В работе [81] отмечается, что «проектирование хорошего языка неизменно представляет собой компромисс между полной универсальностью и тем, что может быть сделано с приемлемой эффективностью. Эффективность, разумеется, определяется по отношению к целевому интерпретатору — обычно являющемуся конкретной ЭВМ. Отсюда следует, что разработка языка сильно зависит от архитектуры аппаратных средств».

Следствием ограничений является то, что в ряде сложных приложений оказывается невозможным применять ЯВУ в качестве универсального средства программирования. Приходится адаптировать алгоритм к аппаратуре и использовать Ассемблер как единственный универсальный инструмент управления данной машиной.

3. Третью проблему можно рассматривать как следствие первых двух. Она состоит в снижении надежности ЯВУ и/или его реализации. В первом случае непосредственно в язык вносятся элементы ненадежного ассемблерного программирования (отключение контроля типов, управление назначением адресов, выход в Ассемблер и т. д.). Это делается, чтобы дать средства управления конкретной аппаратурой, позволяющие, во-первых, повысить эффективность программ и, во-вторых, «обойти» ограничения языка. Разумеется, при этом надо соответствующим образом адаптировать алгоритм к аппаратуре или особенностям реализации. Во втором случае аналогичные средства дает конкретная реализация. Например, в реализациях Фортрана, как правило, отсутствует контроль границ массивов и общих областей. В целом надо отметить основной недостаток этих приемов — снижение надежности программ и усложнение отладки, т. е. потеря важных преимуществ программирования на ЯВУ.

4. Четвертая проблема — эта слабая системная поддержка полного цикла создания программного продукта на ЯВУ, что затрудняет

применение языков. Выше отмечалось, что при разработке БПК значительная часть трудозатрат приходится на этапы отладки, на обнаружение и исправление ошибок при сопровождении и модернизации. Но в традиционных системах именно на этих этапах почти не удается использовать преимущества программирования на ЯВУ:

— из-за принятой в некоторых системах стратегии статического распределения ресурсов необходимо применять сложные ассемблерные методы регулировки объема кода и размещения его в памяти;

— чтобы произвести подробный анализ ошибок при отладке (без специального изменения и повторного выполнения программы), приходится пользоваться не исходным текстом на ЯВУ, а ассемблерной программой, полученной после компиляции с ЯВУ в коды машины.

Иными словами, на этапе отладки программист возвращается на уровень работы с Ассемблером. В целом оказывается, что преимущества, полученные от использования ЯВУ на этапе написания программы, могут незначительно сократить весь объем затрат, связанных с созданием конечного продукта, так как на других трудоемких этапах сохраняются старые методы работы с программой. Если учесть, что ценой за использование ЯВУ является снижение эффективности программы, то становится ясно, почему в традиционных системах наиболее ответственные БПК пишутся на Ассемблере, а ЯВУ используется в основном как средство документирования ассемблерных программ. Резюмируя, можно сказать, что исходной причиной того, что ЯВУ часто проигрывает в соревновании с Ассемблером, является упоминавшийся семантический разрыв между архитектурой и ЯВУ.

Новые возможности. В настоящее время появился ряд факторов, которые изменяют соотношение между вновь разрабатываемыми архитектурами и ЯВУ.

Во-первых, возрос объем знаний о тех механизмах ЯВУ, которые являются наиболее существенными и требуют аппаратной поддержки. Степень аппаратной реализации базовых механизмов ЯВУ может зависеть от класса машины и соответственно допустимого объема оборудования. Например, механизм полного переключения глобального окружения (или контекста) при вызове процедуры одного пакета из процедуры другого пакета может быть реализован полностью аппаратно в случае высокопроизводительных машин и программно — в случае микропроцессора. Но часто встречающийся в программах вызов локальной или глобальной процедуры текущего пакета, не требующий полного переключения контекста, и механизм адресации к переменным нескольких глобальных уровней (а не одного уровня, как в Фортране) целесообразно в обоих случаях реализовать аппаратно. Важно, что знание о том, какие черты ЯВУ

наиболее полезны, сегодня уже может являться отправной точкой разработки системы команд и структуры машины.

Во-вторых, появились значительные продвижения в теории и практике компиляции. Это позволяет при решении задачи эффективной реализации ЯВУ гибко распределить нагрузку между компилятором и аппаратурой так, чтобы удовлетворить внешним требованиям к объему оборудования и/или производительности машины.

Если эти два фактора рассмотреть в сочетании с прогрессом в области разработки элементной базы, заключающимся в снижении стоимости интегральных схем, повышении их быстродействия и степени интеграции, то можно заключить, что в настоящее время появилась возможность в значительной мере ликвидировать рассогласование между возможностями аппаратуры и механизмами ЯВУ. Это позволяет строить ЭВМ, ориентированные на ЯВУ и покрывающие широкий диапазон на шкале производительности.

Языковая проблематика разработки системы Эльбрус. Сказанное во многом определяет исходные позиции разработки универсальной высокопроизводительной системы Эльбрус, т. е. многопроцессорных вычислительных комплексов (МВК) Эльбрус и их общего системного программного обеспечения (ОСПО). Одна из основных целей разработки системы заключалась в том, чтобы за счет полного перевода профессионального программирования на языки высокого уровня упростить создание БПК различного назначения, включая операционную систему комплекса и ОСПО в целом. Причем это надо было сделать без потери эффективности использования ресурсов машины, т. е. памяти и скорости МВК.

В случае системы Эльбрус эта цель достигалась путем приспособления архитектуры к языкам, т. е. включением в число улучшаемых характеристик системы также и архитектуры машины. Эта новая особенность самого метода построения системы привела к тому, что появилась и новая, языковая проблематика разработки вычислительной системы.

Основные вопросы концентрируются вокруг двух проблем. Первая, и, пожалуй, ключевая проблема — это проблема выбора базового языка программирования. Вторая проблема — разработка средств построения систем программирования (СП), удобных для разработки БПК на ЯВУ. Возможны разные подходы к выбору базового языка. Отложим подробный анализ до § 15 и рассмотрим вкратце один пример.

Известен опыт разработки серии машин с привязкой аппаратуры к существующему языку, к языку Алгол. Конечно, для первых шагов в новом архитектурном направлении такое решение было вполне оправдано. Но в результате оказалось, что возможностей Алгола недостаточно, чтобы покрыть нужды программирования систем [82]. Пришлось ввести существенное расширение Алгола и еще че-

тыре его модификации, т. е. всего пять, на первый взгляд, схожих языков. Причем языки соединены, так сказать, «перемычками», чтобы при написании одного программного комплекса можно было пользоваться несколькими языками. Это создало определенную громоздкость языкового обеспечения системы. Чтобы в нем правильно ориентироваться, надо хорошо знать особенности системы.

Из этого примера можно извлечь по крайней мере два полезных вывода. Во-первых, целесообразно делать единый язык для программирования систем в достаточно широком диапазоне приложений. Функциональные возможности такого языка должны быть шире, чем у традиционных ЯВУ. В первую очередь он должен быть более динамичным, но динамика должна быть охвачена контролем правильности обработки объектов. Тогда один и тот же язык может применять и обычный пользователь, и разработчик операционной системы. Но чтобы такой язык был эффективным, требуется аппаратная поддержка.

Второй вывод, концептуально менее весомый, но важный с практической точки зрения, состоит в следующем. Если в язык «погрузить» средства управления файлами и архивом, то не придется делать специальный язык для программирования систем управления базами данных и особый язык управления заданиями. С другой стороны, у языковых средств, которые требуются в этих разделах программирования, нет какой-то особой прикладной специфики. Они не меньше нужны разработчикам транслирующих систем и других программных комплексов, т. е. такие средства целесообразно дать в едином универсальном языке программирования. Но чтобы он эффективно использовал возможности вычислительного комплекса, требуется соответствующая поддержка со стороны операционной системы.

Можно еще привести ряд похожих соображений по поводу характеристик языка (см. § 10). Если их просуммировать, то оказывается, что язык надо делать заново и совместно с архитектурой машины и операционной системы.

Именно такой подход к выбору базового языка был предпринят при разработке системы Эльбрус. За основу взят не существующий язык, а фундаментальные механизмы языков высокого уровня (см. § 2). Для расширения возможностей языка эти механизмы обобщены и разработаны новые механизмы (см. § 3—10). Чтобы отделить первичные механизмы языка от вторичных, получающихся сочетанием первичных механизмов, целесообразно предварительно построить гипотетический язык с ортогональными средствами. Это построение отражено в § 3—9.

На этом базисе был разработан новый язык Эль-76. Его отличительная особенность в том, что он действительно позволил в рамках Эльбруса полностью перевести программирование всех без исклю-

чения систем на язык высокого уровня. Причем во всех приложениях обеспечивается эффективность рабочих программ в сочетании с использованием основных полезных свойств языков высокого уровня (крупноблочность конструкций языка, контроль правильности вычислений, защита программ и данных, виртуализация ресурсов), т. е. именно те свойства, которые упрощают программирование по сравнению с языком Ассемблера традиционных машин. Коротко перечислим основные характеристики языка.

Эффективность программ. Уровень архитектуры Эльбруса поднят к уровню языковых конструкций. Благодаря этому решена главная проблема, из-за которой ограничено применение алгоритмических языков. А именно, исключен эффект расширения кода программы, оттранслированной с языка, по сравнению с программой, написанной вручную в командах. Т. е. Эль-76 позволяет полностью использовать возможности машины. Однако он не является символьным аналогом системы команд не только по своим изобразительным средствам (это видно из примеров § 20 гл. 2), но и потому, что между языком Эль-76 и машинным языком оставлен, если так можно сказать, небольшой «зазор», который не влияет на эффективность, но дает возможность переносить программы, написанные на Эль-76 с одной модели семейства МВК Эльбрус на другую, несмотря на различия в системах команд. Конкретно, Эльбрус-1 и Эльбрус-2 при общности архитектурных принципов отличаются по системе команд. А системное программное обеспечение этих машин, включая операционную систему, полностью написано на Эль-76, и благодаря этому является единым.

Динамизм средств обработки данных. В предыдущих разработках языков программирования объективно считалось, что язык может быть либо динамичным в части управления типами и ресурсами, но неэффективным, либо динамичным и эффективным, но незащищенным в смысле отсутствия или возможности отключения контроля типов и доступности данных. Например, программа, написанная на таком языке, может в процессе работы изменять локальные, т. е. логически не доступные ей данные другой программы. Это происходит из-за наличия в языках так называемых неконтролируемых конструкций (см. § 12). Но такие средства приходится давать и в условиях традиционных архитектур — по-видимому, нет другой практичной альтернативы; т. е. для достижения эффективности и динамичности приходится снижать степень надежности языка.

Это замечание не умаляет полезности статического контроля, но подчеркивает то обстоятельство, что если в области статики на сегодняшний день есть существенные продвижения, то в области динамики на базе традиционных архитектур, т. е. без специальной аппаратной поддержки, трудно было получить язык одновременно эффективный, динамичный и защищенный. В случае Эль-76 за счет

совместной разработки языка и архитектуры Эльбруса удалось эти три свойства соединить в одном языке.

В частности, в языке даны: динамическое управление стандартными и определяемыми типами, динамическое управление временем жизни и размерами объектов, средства динамического сочленения программного комплекса с возможностью выбора вариантов реализации тех или иных модулей. Причем все эти механизмы охвачены динамическим контролем. Вместе с тем программист может регулировать степень динамичности управления типами в зависимости от специфики программы — от полностью статического контроля типов в случае программ (или фрагментов программ), где типы обрабатываемых данных статически известны, до полностью динамического контроля типов, если это соответствует природе программируемого алгоритма.

Обработка ошибок программ. Динамический контроль имеет еще одно применение. Ошибочная ситуация может быть обнаружена во время работы отлаженной программы в реальных условиях. Например, пользователь операционной системы может неверно задать входные данные. Система должна продолжать работать и в этом случае. Для программирования таких «помехоустойчивых» программных систем в Эль-76 разработан механизм исключительных ситуаций, который позволяет обработать ошибку, не прекращая работы системы. Принципы разработки этого механизма изложены в работах [1, 22, 23].

Средства обработки файлов. Совместно с языком были разработаны спецификации средств управления простыми (неструктурированными) файлами и универсальным архивом системы Эльбрус. Такой подход оказался полезным как с точки зрения разработки системы, так и с позиций разработки языка. С точки зрения разработки системы это было полезно, потому что ряд удобных в программировании черт ЯВУ был обобщен для случая обработки архивов. Например, по образцу языковой межмодульной защиты устроена надежная защита личных архивов программ и пользователей; типизация архивных объектов обеспечила возможность контроля правильности их обработки.

Для языка это оказалось полезно потому, что за счет, так сказать, «погружения» в язык средств управления файлами и архивом расширилась сфера его применения. Например, он стал выполнять роль инструментального языка для разработки систем управления базами данных, роль языка управления заданиями. Т. е. удалось и расширить возможности языка, и сократить языковое обеспечение системы, что в конечном счете облегчает работу пользователя.

Подытоживая, можно сказать, что основной особенностью языка, отличающей его от других языков программирования, является сочетание четырех характеристик: высокий уровень програм-

мирования, эффективность рабочих программ, доступ ко всем возможностям аппаратуры и ОС, расширенные функциональные возможности.

Вместе с тем ясно, что новые возможности языка, и в особенности динамичность, опираются на аппаратно-программную поддержку МВК и ОС Эльбрус. Поэтому не преследовалась цель достичь эффективной переносимости языка на архитектуры предыдущего поколения. Однако программная преемственность системы Эльбрус по отношению к существующим машинам также важна. Следовательно, наряду с созданием нового языка надо было обеспечить эффективную реализацию существующих ЯВУ и переносимость программ, написанных на этих языках, в систему Эльбрус. В связи с этим целесообразно в целом рассмотреть языковую стратегию разработки и развития системы Эльбрус.

Применение языков программирования в системе Эльбрус. Выделим такие задачи, стоящие перед разработкой средств профессионального программирования: переносимость программ, высокий уровень программирования, расширение сферы применения ЯВУ, эффективность рабочих программ. Решить эти задачи в рамках одного языка трудно, не жертвуя при этом или переносимостью, или надежностью, или эффективностью, или универсальностью. Поэтому выработаны два языковых направления.

Первое направление является «эльбрусовской» линией развития языков. В настоящее время оно ориентировано на решение задачи разработки эффективного универсально применимого (в такой же мере, как Ассемблер в сочетании с языком управления ОС) языка программирования систем, сохраняющего во всех приложениях преимущества ЯВУ. Это направление принципиально основывается на том, что для решения задачи необходима совместная разработка языка и архитектуры. Поскольку такой ЯВУ ориентирован на возможности нового класса архитектур, он не является эффективно переносимым вниз, т. е. на традиционные ЭВМ.

Отметим, что аналогичное архитектурное направление развивается в других работах по новым ЭВМ различной производительности вплоть до класса микропроцессоров (например, микропроцессоры SPUR [83], SOAR [84]), а также имеется близкий аналог в проекте ЭВМ пятого поколения в виде Новой машины Неймана (см. § 14). Поэтому можно предполагать появление нескольких машин в данном классе архитектур, продолжение эльбрусовского языкового направления и как следствие — переносимость в рамках машин данного класса программ на языке, использующем возможности новой архитектуры.

Второе языковое направление ориентировано на решение задачи обеспечения совместимости системы Эльбрус с

существующими ЭВМ на распространенных ЯВУ (Алгол, Фортран ПЛ/1, Паскаль) и их диалектах. В рамках этого же направления осуществляется реализация экспериментальных ЯВУ и СП следующего поколения.

Два названных направления достаточно хорошо согласованы между собой. Действительно, первое направление предполагает выделение и системную реализацию базовых механизмов ЯВУ. Поэтому, во-первых, при реализации семантических механизмов традиционных языков можно использовать готовые системные средства. Во-вторых, за счет стандартизации на системном уровне методов реализации схожих языковых механизмов упрощается стыковка модулей, написанных на разных языках. Это особенно важно с точки зрения использования пакетов прикладных программ, написанных на разных языках. Объединяет оба направления и то, что разработаны единые системные технологические средства, облегчающие создание развитых СП, поддерживающих полный цикл программирования на ЯВУ.

В рамках второго направления в системе Эльбрус реализован широкий спектр языков. В Новосибирском филиале ИТМ и ВТ реализованы [37—43]: два транслятора языка Алгол-60, причем второй транслятор поддерживает диалекты Алгола-60, используемые на БЭСМ-6; два транслятора языка Фортран, один из которых поддерживает диалекты Фортрана, используемые на БЭСМ-6 и ЕС ЭВМ, и Фортран-77; трансляторы языка ПЛ/1, базис которого совместим с реализацией ПЛ/1 на ЕС ЭВМ, а также языков Кобол, Си, Ада. Серия компилирующих и интерпретирующих систем реализована в ЛГУ [44—50]: Паскаль, Клу, Лисп, АБВ, Рефал, диалоговый Алгол, Снобол. Кроме того, реализованы трансляторы языков Алгол-68 (ИТМ и ВТ, НФ ИТМ и ВТ) [51] и Симула-67 (РГУ) [52]. В ИТМ и ВТ реализованы транслятор языка Эль-76, многоязыковые инструментальные компоненты СП Эльбрус (система символьной аварийной выдачи и отладки программ, комплексатор программ, средства выдачи структурной документации о программе), пакеты системной поддержки компонентов СП (см. § 13). СП Эльбрус является многоязыковой СП в том смысле, что, во-первых, она обслуживается единым инструментальными компонентами и, во-вторых, из программы, написанной на одном языке, можно вызывать программу, написанную на другом языке.

Особенности разработки системы программирования. Создание системы программирования, достаточно удобной для пользователя, работающего на ЯВУ, является сложной и трудоемкой задачей. Реализация транслятора с языка — это иногда меньшая часть всего объема работ; остальное — это разработка динамической поддержки, архива и инструментальных компонентов, которые обеспечивают отладку и другие сервисные возможности. Из-за объемности этой

работы она не всегда доводится до конца, и пользователь получает транслятор без достаточного сервиса.

Отличительная особенность системы Эльбрус в этом вопросе заключается в том, что была применена новая технология разработки СП. Цель этой технологии состоит в том, чтобы упростить создание СП, взаимодействующей с пользователем в терминах исходной программы на ЯВУ, а ее существо заключается в следующем:

— ряд механизмов, необходимых для систем программирования, заранее был встроен в архитектуру машины и ОС. Кроме того, внутреннее представление программы устроено так, чтобы в нем сохранялась информация об исходной статической структуре программы и структуре процесса ее выполнения. Эта информация сохраняется в терминах языка высокого уровня (см. § 13), причем система так организована, что символьная аварийная выдача не требует ни специальной подготовки и повторного запуска программы, ни дополнительных ресурсов оперативной памяти и производительности расходуемых в процессе нормальной работы программы. Если при работе программы в реальных условиях зарегистрирована ошибка, то в терминах исходного текста программы и ЯВУ выдается информация о текущем состоянии программы и сообщение об ошибке;

— существенная часть работы по системе программирования один раз выполнена централизованным образом в ОСПО Эльбрус. Это — упоминавшиеся выше многоязыковые инструментальные компоненты системы программирования;

— разработчикам трансляторов даны средства подключения новых трансляторов к этому готовому общесистемному сервису. Например, первоначально многоязыковая система динамической диагностики использовалась для языков Эль-76, Алгол и Фортран. Затем, когда в ЛГУ был реализован транслятор языка Паскаль, для него уже не надо было разрабатывать, например, новой системы символьной аварийной выдачи. Транслятор подключился к многоязыковой системе программирования, и пользователи одновременно с транслятором получили средства отладки программ.

Такой подход к разработке СП упростил оснащение трансляторов, а значит, и пользователей сервисными возможностями программирования на ЯВУ.

Следующее поколение методов программирования. В отличие от операционного стиля программирования, эти методы предполагают, что пользователь вычислительной системы, не являющийся профессиональным программистом, будет в основном определять, «что» надо сделать, а не детали того, «как» это делать. Такой подход соответствует современной тенденции к информатизации общества, накоплению знаний в ЭВМ и разработке средств, упрощающих использование накопленных знаний, в том числе пакетов прикладных программ (ППП). Значительные теоретические и практические

шаги в этом направлении сделаны в отечественных работах по входным языкам: Утопист системы Приз [58], Декарт системы Спора [59], по экспертным и схожим системам.

Данное языковое направление дает новый импульс развитию архитектур ЭВМ, так как здесь нужны особые способы обработки информации: поиск по образцу, перебор вариантов, работа с множествами. Откладывая обсуждение проектов таких архитектур до § 14, отметим один аспект, важный с точки зрения данной работы. Исходя из общих соображений и судя по набору архитектур в перспективных проектах ЭВМ пятого поколения, процедурные ЯВУ и соответствующие им архитектуры по-прежнему сохранят свою важную роль, во-первых, в практическом отношении — как инструменты профессионального программирования систем, а во-вторых, в теоретическом отношении — как архитектуры, где отрабатываются приемы реализации фундаментальных механизмов, общих для процедурных ЯВУ и языков следующего поколения.

Можно предположить, что переход к широкому практическому внедрению методов программирования следующего поколения будет носить не революционный характер — от языка Ассемблера сразу к непроцедурным языкам, а характер эволюционного процесса, важным этапом которого является полный перевод программирования на процедурные ЯВУ. Кроме того, в перспективе можно предполагать сочетание процедурных и непроцедурных элементов стиля программирования, по крайней мере в области профессионального программирования.

В данной работе, говоря о ЯВУ, в основном имеются в виду многоцелевые процедурные ЯВУ, предназначенные для программирования систем, и соответствующие им архитектуры ЭВМ. Задача разработки систем поддержки непроцедурного (спецификационного, функционального) программирования рассматривается как задача разработки сложных систем с помощью процедурного ЯВУ.

2. Фундаментальные черты языков высокого уровня

Основные преимущества ЯВУ перед языком Ассемблера традиционных машин обусловлены тем, что в ЯВУ был введен ряд приемов, используемых в математике при конструктивном описании алгоритмов и в других построениях. Рассмотрим эти приемы.

2.1. Типизация объектов. Этот прием позволяет обеспечить смысловую различимость объектов, разбивая их на множества и определяя для множества объектов набор допустимых операций. Вопрос о структуре объектов рассматривается ниже, так как считается, что структура объекта — это характеристика некоторого конкретного множества. Здесь пока важно, что в принципе есть возможность разбивать объекты на множества и устанавливать

принадлежность объекта тому или иному множеству. Используя различимость типов (и ряд вводимых ниже механизмов), можно определять набор операций, допустимых для объектов некоторого типа и недопустимых для других объектов. В отличие от этого в языке Ассемблера объекты неразличимы по смыслу, т. е. одну и ту же информацию можно обрабатывать любой операцией и в зависимости от применяемой операции трактовать информацию в том или ином смысле.

Другой стороной типизации является семантический контроль правильности обработки. Различая типы объектов, можно установить, что к объекту применяется операция, недопустимая для данного типа. Если не различать типов, то соответственно нельзя осуществить этот вид контроля.

2.2. Структурирование объектов. Как прием преодоления сложности описания алгоритма, в ЯВУ используются средства композиции сложного из простого: средство определения новых типов составных объектов, построенных из других объектов, как правило, ранее определенных типов, а также средство определения с помощью процедурной композиции сложных действий из простых [60].

Особенностью ЯВУ является то, что осуществляется контроль правильности обработки объекта в соответствии с его логической структурой *). Объект можно декомпозировать на логически составляющие элементы-объекты и обрабатывать их независимо, т. е. обработка одного элемента не затрагивает другого элемента. В отличие от этого программа на языке Ассемблера может обрабатывать память как смежную последовательность битов или других физических элементов памяти, не различая границ объектов и их логических компонентов. В случае процедуры, в принципе, возможна передача управления снаружи в любую точку ее тела (а в ЯВУ процедура выполняется как неделимое действие). Таким образом, в общем случае обработка информации ассемблерной программой ведется без контроля логической структуры объектов.

2.3. Контекст. Известный прием, используемый при описании алгоритмов состоит в том, что сначала вводится и объясняется некоторый набор терминов и затем описывается алгоритм, понимаемый в контексте этого набора терминов. В результате появляется возможность обеспечить различимость контекстов, отличающихся по совокупности терминов и по смыслу одинаково обозначенных терминов. Если алгоритм трактуется всегда в одном и том же контексте, то это обеспечивает однозначное понимание алгоритма в тех случаях, когда он сам используется как элементарное

*) Этот аспект можно, конечно, отнести к контролю типов. Однако ввиду его практической важности он выделяется особо.

понятие в контексте, определяемом для нового алгоритма. При этом оказывается неважным, какие в этом контексте вводятся обозначения для других терминов. Аналогично в ЯВУ алгоритм обычно жестко связывается с контекстом обозначений, с помощью которых в описании алгоритма упоминаются объекты, введенные в этом контексте (типы и операции, обозначаемые с помощью изображений процедур, также будем относить к объектам, подробно см. ниже).

Некоторые объекты, такие как числа, имеют общепринятые обозначения. Относительно них можно считать, что они понимаются в стандартном (или предопределенном) контексте. С помощью описания с объектом можно связать произвольное обозначение. В описании задаются обозначение и обозначаемый объект. Совокупность обозначений, описанных вне процедуры и действующих внутри нее, называется глобальным контекстом обозначений для данной процедуры. Глобальный контекст обозначений является аналогом контекста терминов, которые могут быть общими для нескольких алгоритмов.

В теле процедуры могут находиться локальные описания. Они задают локальный контекст процедуры (аналог набора терминов, используемых только для описания одного конкретного алгоритма). Описания формальных параметров задают аргументы операции. С помощью механизма параметров в контекст процедуры при вызове могут быть переданы объекты из других контекстов.

Совокупность из контекста обозначений, обозначаемых объектов, доступных элементов (элементов элементов и т. д.) этих объектов и объектов, косвенно доступных через посредство объектов типа имя (см. ниже), будем называть просто контекстом. В результате жесткой привязки контекста к процедуре, во-первых, не происходит смысловой «путаницы» в случае, когда в разных контекстах есть одинаковые обозначения для разных объектов, и во-вторых, обеспечивается возможность ограничить контекст и выявлять случаи, когда в описании алгоритма используются «посторонние» обозначения, не определенные для данного алгоритма. В отличие от этого в случае Ассемблера нет возможности ограничить контекст. Здесь используется общий контекст для различных алгоритмов (или различных частей одного алгоритма), что может приводить к неверному в смысловом отношении использованию обозначений.

Поскольку для реализации объектов используется память и есть операции, изменяющие содержимое памяти, то указанное свойство приводит к тому, что программа может изменить содержимое логически недоступной части памяти, например, из-за содержавшейся в программе ошибки. Обычно это называют нарушением контекстной защиты. Нарушение защиты может по-

влечь за собой непредсказуемое поведение программ, для которых данная часть памяти является логически доступной.

Основной недостаток нарушения защиты состоит в том, что трудно выяснить изначальную причину неправильной работы программ. В этом отношении преимущество ЯВУ состоит в том, что он обеспечивает контекстную защиту. Чтобы отличать механизм управления контекстом, принятый в ЯВУ, от общего контекста программ языка Ассемблера, первый будем называть механизмом р а с п р е д е л е н н о г о к о н т е к с т а *). При этом имеется в виду, что вся совокупность существующих в некоторый момент объектов разбита на, возможно, частично пересекающиеся, контексты, закрепленные за отдельными процедурами.

Заметим, что в некоторых ЯВУ используется механизм, близкий к механизму общего контекста, что можно отнести к недостатку этих языков. Пример нарушения защиты в языке Лисп, приведенный автором этого языка, состоит в следующем. Пусть в функции *а* есть обращение к глобальному идентификатору *x*, а в функции *в* есть обращение к глобальному идентификатору *x*, а в функции *в* есть локальный объект *x*:

а = (*лямбда* (. . .). . .*x*. . .)
в = (*лямбда* (*y*) (*прог* (*x*). . .(*y*. . .). . .))

Если произвести вызов функции *в*, передав ей *а* в качестве фактического параметра, то в результате выполнения вызова формальной функции (*y*. . .) процедура *а* получит доступ к локальному объекту *x* процедуры *в*, т. е. к объекту, который не был явным образом доверен процедуре *а*.

Перечисленные выше фундаментальные черты алгоритмического языка (типизация объектов, композиция/декомпозиция объектов, механизм распределенного контекста) в этой работе рассматриваются как основные отличия ЯВУ от языка низкого уровня. Соответствующий этим чертам семантический контроль обусловливает преимущества применения ЯВУ по сравнению с языком Ассемблера.

2.4. Дополнительные термины. Поясним ряд вторичных терминов, которым придается разная трактовка в языках.

Объекты. Языки, вообще говоря, отличаются по набору стандартных типов объектов и средств композиции. В большинстве процедурных ЯВУ различаются следующие классы типов: стандартные скалярные (упорядоченные) типы **), типы имен, типы процедур,

*) Термин «статический контекст» не используется, так как в работе будет рассмотрен случай, когда в процессе выполнения процедуры в ее глобальном контексте могут динамически появляться новые обозначения.

**) В некоторых языках существует возможность определять новые скалярные типы со стандартными операциями сравнения и следования.

типы составных объектов (структур и массивов). Для каждого класса определены стандартные операции: стандартные операции для скаляров (сравнение, арифметические и логические операции и проч.), присваивание и разыменование для имен, операция вызова для процедур, выбор элемента для составных объектов. Некоторые операции, например, арифметические операции определены для нескольких возможных типов аргументов и независимо от типа обозначаются одинаково. Это свойство операций называют полиморфизмом.

Относительно объекта типа имени говорят, что он обладает свойством именовать (ссылаться на) другой объект. Совокупность из имени и именуемого объекта образует переменную. Второй объект этой совокупности называется значением переменной. Присваивание имени производит замену значения переменной, а результатом разыменования имени является текущее значение переменной.

В отношении процедур будем, как в Алголе-68, различать исполнение конструкции «изображение процедуры» и исполнение конструкции «вызов». Под термином «процедурный объект» понимается объект, получаемый в результате исполнения первой конструкции. Этот объект, который можно назвать экземпляром изображенной процедуры, представляет собой совокупность из исходного изображения и текущего контекста, глобального по отношению к процедуре. Таким образом, глобальный контекст закрепляется за процедурным объектом в момент его создания.

В отношении составных объектов будем считать, что такой объект состоит из элементов, каждый из которых в свою очередь является объектом (скаляром, именем, процедурой, составным) и снабжен обозначением, т. е. идентификатором в случае структур и индексом — в случае массивов.

В дальнейшем в вопросе выбора классов типов для языка с динамическим управлением абстрактными типами данных (ДАТД) мы будем следовать рассмотренной традиции процедурных языков, сделав ряд уточнений, соответствующих структуре языка ДАТД. Надо отметить, что это не единственный вариант выбора классов типов.

В базис языка Лисп входят только два типа — атом и список, только один способ композиции — формирование списков, и есть функция интерпретации списка как действий. В случае ДАТД такой вариант выбора базиса не используется, так как при построении этого языка не преследуется цель дать средства обработки процедуры как структуры данных. Такие средства будут введены в последующем изложении, при описании разработки базовых структур системы Эльбрус, используемых системами программирования.

Универсальные средства обработки. Общезначимыми механизмами ЯВУ, т. е. применимыми для всех объектов, можно считать средства обозначения объектов, средства определения времени жизни объекта, присваивание объекта, выдачу объекта в результате разыменования и выполнения процедуры, передачу объекта параметром, включение объекта в составной объект в качестве элемента, проверку тождественности объектов, проверку типа объекта.

Принципом универсальной применимости базовых средств обработки объектов назовем принцип, состоящий в том, что названные механизмы действительно определены для всех объектов, представленных в языке *). Выделение этого момента связано с тем, что этот принцип удовлетворяется не во всех ЯВУ, что ограничивает функциональные возможности языка.

Статика и динамика типов. Языки существенно различаются по средствам управления типами. В основном для контроля типов используются две стратегии: статическая и динамическая. Языки первой разновидности, такие как Алгол-68, Клу, Модула, Эвклид, Тартан, Ада [61, 85—89, 62], будем для краткости называть языками класса С, а языки второй разновидности — языками класса Д (Лисп, РОР-2 [90], Эйлер [85], АПЛ, ИНФ [63]). К числу достоинств языков класса Д относят прежде всего простоту программирования и гибкость, [64,91]. Наряду с этим отмечается, что статический контроль типов, хотя и ограничивает возможности языка, существенно увеличивает скорость выполнения программы.

Существует еще третий класс языков — так называемые бестиповые языки **). Программы, написанные на этих языках, как уже говорилось, осуществляют обработку содержимого поименованных областей памяти, причем смысл информации определяется в зависимости от того, какая над ней выполняется операция. По своим синтаксическим формам бестиповый язык может быть во многом схож с ЯВУ. Но отсутствие семантического контроля сближает эти языки с ассемблерными средствами программирования (АСП). В дальнейшем мы в основном будем рассматривать языки классов С и Д.

Особенности, отличающие языки класса С от языков класса Д, формально состоят в следующем. В определении типа составного

---

*) Оговорка, касающаяся абстрактных типов, состоит в следующем. В соответствии с рассмотренным ниже подходом, если тип является в некотором контексте абстрактным, то нельзя в данном контексте непосредственно выполнить генерацию объекта и проверку типа объекта. В пакете обработки объектов данного АТД должны быть предусмотрены соответствующие примитивы, обеспечивающие опосредованную генерацию и проверку типа.

**) В зависимости от уровня изобразительных средств различают две разновидности бестиповых языков [91]: ассемблерные языки и языки, подобные ЯВУ.

объекта фиксируются типы его элементов, фиксируются типы параметров процедуры и тип результата процедуры функции (отсюда появляется класс разных типов процедур), фиксируется тип объекта, именуемого объектом типа имя (отсюда появляется не один тип имени, а класс разных типов имен). Таким образом, особенностью языка класса С является то, что, кроме основных (с точки зрения данной работы) свойств объектов (например, у имени — именовать любой другой объект, у составного объекта — содержать элементом любой другой объект), в определении типа объекта статически фиксируются дополнительные свойства (именовать или содержать элементом объект статически известного типа). Происхождение этих черт языков класса С будет рассмотрено в § 13.

3. Язык с динамическим управлением абстрактными типами данных

Перейдем к рассмотрению элементов языка с динамическим управлением абстрактными типами данных (АТД). Для краткости будем называть этот язык ДАТД. Сначала надо определить, как понимается термин АТД.

3.1. Механизм АТД. В данной работе считается, что первичными являются механизмы, названные в пп. 2.1—2.3. Механизм АТД является вторичным в том смысле, что он получается сочетанием первичных механизмов. Прежде чем описать способ построения механизма АТД, сделаем три уточнения. Во-первых, будем считать, что собственно тип также является объектом типа *тип*. Некоторый тип, как и любой другой объект, может быть доступен (известен) в одном контексте и недоступен (неизвестен, скрыт) в другом контексте. Во-вторых, составной объект можно создать с помощью генератора только в таком контексте, где доступен тип. Ключевой момент состоит в том, что декомпозировать объект, т. е. получить доступ к элементам, можно только в контексте, где доступен тип данного бъекта.

Третье уточнение касается трактовки эквивалентности типов. Будем считать, что два объекта имеют один и тот же тип, если тип первого объекта и тип второго объекта суть один и тот же объект. Объект типа *тип* выдается в результате исполнения изображения типа. По аналогии с семантикой исполнения изображения других объектов целесообразно различать два случая.

В первом случае считается, что изображение обозначает некоторый глобальный объект, определенный в стандартном окружении, и в результате исполнения изображения выдается этот объект. Отсюда вытекает, что два текстуально одинаковых изображения выдают один и тот же объект. Это, например, имеет место для целых чисел. Такой же подход используется в Алголе-68 для всех изображений типов. Он приводит к так называемой «структурной эквива-

лентности» типов, т. е. к тому, что все типы являются безусловно глобально известными и в принципе нет возможности скрыть тип объекта.

Во втором случае считается, что изображение типа действует аналогично генератору объекта типа *тип*; в результате его исполнения выдается новый тип, который тождественно равен (эквивалентен) только сам себе. Этот механизм локальных типов дает два важных практических следствия. Во-первых, он позволяет в смысловом отношении отличать объекты типа m_1 от объектов типа m_2 даже в том случае, если изображения m_1 и m_2 текстуально совпадают. Во-вторых, можно не вводить новых механизмов для обеспечения техники скрытых типов, принципиально используемой механизмом АТД (см. ниже), а воспользоваться локальностью типов и общим механизмом распределенного контекста. Этот подход используется в данной работе.

Абстрактный тип. Предлагаемый способ построения состоит в следующем. Понятие абстрактного типа рассматривается как относительное в том смысле, что в контексте, где тип недоступен, но доступны объекты данного типа, этот тип является абстрактным. Тот же самый тип может быть доступен в другом контексте, тогда относительно последнего он является конкретным. Это назовем принципом контекстной известности типа.

Содержательно «абстрактность» типа заключается в том, что неизвестна структура объектов данного типа. Формально это поддерживается тем, что в данном контексте нельзя декомпозировать объект, его можно обрабатывать только как единое целое с помощью общезначимых средств (передача параметром в процедуру и пр., см. п. 1.4).

В процессе передачи объекта абстрактного типа параметром в процедуру может наступить такой момент, когда объект передан в процедуру, в глобальном контексте которой известен тип данного объекта. В этом контексте тип уже является конкретным; процедура может декомпозировать объект на составные части и произвести необходимую поэлементную обработку объекта.

Отсюда видно, что часто используемое выражение «АТД определяется путем определения пакета операций (процедур) обработки объектов данного АТД» достаточно условно. Формально любая процедура, в контексте которой известен некоторый тип, является в указанном выше смысле операцией обработки объектов этого типа. Вопрос о том, объединяются ли такие процедуры в пакет и, если объединяются, то в какое число пакетов (один или несколько), не имеет отношения к конструированию языка, а является вопросом организации программ.

Пакет операций. Разумеется, язык должен быть устроен так, чтобы можно было описать пакет процедур, особое свойство которых

состоит в том, что их глобальный контекст по крайней мере частично недоступен процедурам, использующим пакет. Но будет ли такой пакет использоваться для организации операций обработки АТД или для других нужд (например, как библиотека тригонометрических функций) — это уже вопрос применения данной возможности языка.

В связи с этим определим второй элемент подхода к организации механизма АТД. Он состоит в том, что с формальной точки зрения в язык не надо вводить специальных конструкций описания пакетов, обладающих отмеченным выше средством. Для этого достаточно фундаментальных механизмов композиции объектов и распределенного контекста. Подробно это утверждение раскрыто в последующих параграфах.

Ограничение прав доступа. Учитывая вышеизложенное, можно сказать, что механизм распределенного контекста можно использовать как механизм ограничения прав доступа к объекту. Рассмотрим диапазон возможных ограничений. В предельном случае объект может быть полностью недоступен в некотором контексте. Если же он доступен, но неизвестен его тип, то недоступны элементы объекта.

Пользуясь этим свойством, можно в принципе устроить программы так, чтобы в разных контекстах имелись различные права доступа к объекту. Возможны такие варианты:

— программируются несколько пакетов операций обработки объекта одного и того же типа (глобального для этих пакетов), обеспечивающие разные права доступа к этим объектам. Затем в контекст процедур, пользующихся разными правами, выдаются соответствующие пакеты (далее процедуры, использующие пакеты обработки АТД, будем для краткости называть пользователями пакетов);

— применяется метод заключения объекта в «оболочку». Для этого вводится дополнительный тип (известный, как и основной тип, в контексте пакета, но не в контексте пользователей). Например, тип оболочки представляет собой тип структуры одним компонентом, которым является объект основного типа, а второй компонент обозначает различные права обработки первого, закодированные целыми числами. Тогда один и тот же объект основного типа можно заключить в оболочки, отличающиеся правами доступа, и выдавать в таком виде в контекст пользователя. Каждая операция пакета устроена так, что она анализирует права доступа и в зависимости от результата анализа выдает сообщение о том, что применение данной операции запрещено, или выполняет необходимую обработку *);

*) Именно таким способом в системе Эльбрус устроена защита файлов, в том числе ограничение прав модификации файла.

— объект выдается в контекст пользователя либо заключенным в оболочку, либо в виде объекта основного типа. Во втором случае к нему применимы все операции пакета, а в первом случае — только некоторые операции в зависимости от того, как заданы права доступа. Возможны и другие способы решения этой задачи в конкретных системах. Важно, чтобы в языке программирования был базовый механизм, позволяющий полностью закрыть доступ к информационному содержимому объектов.

Связь с технологией программирования. Механизм контекстной известности типов является важным элементом поддержки технологии модульного программирования: поскольку гарантируется, что сведения о внутренней организации объектов некоторого типа не проникают в процедуры, для которых этот тип является абстрактным, то можно подменять организацию (или р е а л и з а ц и ю) объектов данного типа, не меняя алгоритмов процедур пользователей. (При этом, разумеется, меняются алгоритмы операций, т. е. процедур, для которых тип является конкретным.) Кроме того, можно одновременно иметь различные варианты реализации одного и того же типа и соответствующие им варианты пакета операций, имеющие одинаковые, с точки зрения пользователя, функциональные возможности. Одна и та же программа без изменения алгоритмов может использовать разные варианты пакетов.

В связи с этим можно отметить некоторую непоследовательность языков класса С с механизмом АТД *). Структура описания переменных и параметров в этих языках требует, чтобы идентификатор типа значения переменной был известен в том контексте, где находится это описание. Это требуется и в том случае, когда формально тип объекта, являющегося значением переменной, неизвестен в контексте. Иначе нельзя осуществить статический контроль типов. Во-первых, в этом заключено определенное противоречие: обозначение объекта (т. е. типа) известно, а сам объект недоступен. В случае других объектов такой особенности нет. Во-вторых, появляется особое описание «приватных» типов, как в языке Ада. В-третьих, в спецификации пакета указано полное описание структуры объектов скрытого типа. Это дает возможность соответствующим образом распределять память при компиляции описания переменной скрытого типа. Если эта особенность языка действительно используется компилятором, то в код программы, использующей пакет, попадает информация о внутренней структуре объектов АТД, и следовательно, подмена реализации АТД требует перекомпиляции всех программ, использующих данный пакет. Ниже будет показа-

*) Имеются в виду языки типа Ада, где пакет существует в одном экземпляре, а в программе могут быть несколько переменных, значениями которых являются объекты АТД.

но, что в случае динамического контроля нет необходимости вносить в язык какие-либо особые конструкции для скрытых типов.

Подходы других языков. Сначала вернемся к вопросу о возможностях Ассемблера. Из изложенного видно, что механизм АТД принципиально использует типизацию и структурирование объектов и механизм распределенного контекста. Поэтому, если в языке отсутствуют эти базовые черты, то достаточно сложно организовать АТД в том смысле, как они понимаются в данной работе. Именно так обстоит дело с традиционным Ассемблером. Можно использовать макросредства, но они дают только лаконичность и, возможно, большую выразительность записи. Но важнейшим, с точки зрения упрощения отладки программ, считается семантический контроль правильности обработки объектов. Этого в макроопределениях обычно нет. Если же набор макроопределений устроен так, что он обеспечивает семантический контроль, то его уже трудно отличить от ЯВУ с механизмом АТД. Однако, по имеющимся сведениям, такая практика не распространена (вместо этого строят новый ЯВУ).

Иногда в ЯВУ вводят ограниченные макросредства, позволяющие на первый взгляд определять новые типы. Если этим средствам, как в языке Блисс [92], не придается механизм контроля, то их тоже нельзя рассматривать как механизм АТД. Локально определяемые типы Паскаля — это только часть механизма АТД, полного контроля нет, так как известна и доступна для обработки внутренняя структура любого составного объекта, находящегося в контексте процедуры, и нет средств организации пакетов.

В отличие от этого в расширении Алгола-68 введены модули для описания пакетов, но здесь мешает другое — безусловная глобальность типов. Всегда можно получить доступ к элементу объекта любого типа, если только «угадать» идентификатор и тип элемента. Чтобы решить эти проблемы, в языках Клу, Ада, Модула, Эвклид вводятся специальные (иногда противоречащие общим концепциям) конструкции для описания скрытых типов и пакетов операций обработки.

Особое место принадлежит языкам, в которых используется концепция удерживания объекта до тех пор, пока существует путь доступа к этому объекту. Примером такого языка является АБВ [65], в котором структура представляет собой совокупность удержанных локальных объектов процедуры, активно представленных в момент возврата из этой процедуры. Такая структура может состоять из процедур, контекстом которых являются символьные имена элементов структуры. Пользуясь этим, можно программировать пакеты обработки АТД.

Другим примером является проект языка Орегано [93], в котором, в отличие от языка АБВ, нельзя описать процедуры в кон-

тексте структуры *). В обоих примерах остается открытым вопрос защиты: нет возможности ограничить права доступа к элементам объекта, выдаваемого в контекст пользователя, т. е. тип объекта всегда является конкретным, и пользователь может изменить информационное содержимое любого объекта.

Концепция АТД в сочетании с динамическим управлением типами в настоящее время недостаточно полно проработана. В основном остается открытым вопрос о защите объектов скрытых типов. В статических языках этот вопрос решается достаточно просто, поскольку во время компиляции можно определить, какие типы являются скрытыми. Соответствующим образом устроены языки этого класса (Клу, Ада). Что же касается динамических языков, то их особая структура требует новых решений данного вопроса.

Применение известных методов не позволяет обеспечить ограничение прав доступа к объекту. Например, в языках типа EL/1 [94], содержащих в том или ином виде средства определения новых типов и частично динамику типов, конструкция выбора элемента структуры осталась той же, что и в статических языках**), и позволяет получить доступ к элементу любого объекта, находящегося в текущем контексте, независимо от того, где описан тип объекта. Т. е. в части обеспечения защиты результат даже уступает языкам класса С, где известность типа можно проконтролировать во время компиляции, чтобы запретить доступ к элементам объектов скрытого типа.

3.2. Цели конструирования языка ДАТД. При разработке ДАТД решалась задача построения базовых элементов гипотетического или идеализированного языка, в котором отображены названные выше фундаментальные механизмы ЯВУ, и на основе их сочетания получен механизм АТД. В связи с этим можно выделить такие конкретные цели:

1) Первая цель состоит в том, чтобы в качестве ядра ДАТД ввести язык, особенностью которого является сочетание динамического управления типами с механизмом АТД, понимаемом в том смысле, как определено выше.

2) Вторая цель состоит в том, чтобы ввести в механизм распределенного контекста возможность более гибкого управления кон-

---

*) Сопоставление этих языков чисто формальное. Поскольку разработки преследовали разные цели, то и результаты надо оценивать по-разному.

**) Есть существенная разница в эффективности реализации выбора элемента. В языках класса С смещение элемента относительно начала объекта известно при компиляции, а в языках типа EL/1 необходимо производить ассоциативный поиск в процессе выполнения программы.

текстом и временем жизни процедурных объектов, чем это имеет место в распространенных языках. Это позволит программировать пакеты обработки АТД с помощью структур и процедур, не используя каких-либо специальных конструкций. В связи с этим введем несколько определений. Если изображение процедуры *в* непосредственно текстуально вложено в изображение объекта *а* (или, другими словами, объект *а* является ближайшим текстуально охватывающим процедуру *в*), то будем говорить, что процедура *в* непосредственно структурно подчинена *а*. Кроме того, будем говорить, что процедура *в* контекстно подчинена объекту *а*, если в *в* действуют идентификаторы, локально описанные в *а*. В языках типа Алгол-60 и Паскаль имеют место следующие ограничения: процедура может быть контекстно и структурно подчинена только процедуре, но не структуре (составному объекту), а контекстная подчиненность совпадает со структурной. Это приводит к двум следствиям: если процедура *в* непосредственно вложена в *а*, то в *в* действуют идентификаторы, описанные в *а*, и чтобы в *в* действовали идентификаторы, описанные в *а*, нужно текстуально вложить *в* внутрь *а*.

Чтобы расширить возможности языка, надо допустить включение процедур в состав структур, допустить, чтобы процедура могла быть контекстно подчинена структуре, и разрешить задавать контекстную подчиненность независимо от структурной подчиненности. Тогда процедуру генерации пакета процедур обработки АТД можно программировать, например, так. Описывается функция, производящая следующие действия:

— динамическая генерация структуры скрытой части пакета (назовем ее А), куда как составные части входят структура, составленная из интерфейсных процедур (назовем ее В), а также скрытые элементы, включая локальное описание внутреннего представления объекта АТД (т. е. просто описание некоторого типа *m*);

— значением функции является структура В. Основная структура А также сохраняется по завершении выполнения функции. При этом она оказывается скрытой от пользователя пакета, но доступной интерфейсным процедурам. Таким образом, основная функциональная особенность процедуры генерации состоит в том, что она выдает не весь объект А, но только его часть В. Тип *m* является элементом скрытой части, т. е. является скрытым типом, вследствие чего процедуры могут выдавать объекты типа *m* в контекст пользователя, но последний ничего не может с ними сделать, кроме как вернуть этот объект для обработки в какую-либо процедуру пакета.

Если же попытаться пойти по этому пути при наличии в языке названных выше ограничений, то возникают следующие трудности:

— нет средств, позволяющих вводить пакет процедур и других определений как независимый составной объект и определять процедуры, использующие такие пакеты;

— время жизни процедурного объекта $p1$ равно времени выполнения ближайшей охватывающей процедуры $p2$, так как в $p1$ действуют локальные описания процедуры $p2$. Поэтому $p2$ не может выдать $p1$ в качестве результата выполнения. В рассмотренном алгоритме имеет место именно этот случай, так как все процедуры, вложенные в объекты А и В, опосредованно структурно подчинены процедуре генерации пакета.

3) Третья цель конструирования данного языка состоит в том, чтобы определить подход к разработке средств гибкого управления степенью динамики контроля типов, допускающего в пределе переход к статическому контролю. С практической точки зрения это позволяет использовать статический контроль только в тех разделах программ, где статика типов соответствует природе описываемого алгоритма. Для данной работы подобный анализ важен потому, что, прослеживая процесс перерастания динамического контроля в статический, можно, во-первых, выяснить, чем конкретно ограничена возможность описания алгоритмов на языке класса С по сравнению с языком класса Д, и, во-вторых, определить способ «дозированного» внесения статики в язык класса Д.

3.3. Последовательность изложения. Конструирование языка ДАТД будет проведено следующим образом:

1) Чтобы достичь первой цели, имена, структуры и процедуры вводятся в духе языка класса Д.

2) Чтобы достичь второй цели, механизм распределенного контекста описывается в терминах структур. Для этого вводятся два основных класса объектов: структуры данных и для обозначения действий над данными,— процедуры без локальных описаний. Структуры могут включать в качестве элементов процедуры, а процедуры выполняются в глобальном контексте, составленном из элементов структур. Затем в качестве сокращенной записи для одного частного случая вводятся процедуры с локальными описаниями.

3) Вводятся абстрактные типы путем применения механизма распределенного контекста для именования типов.

4) Приводится пример определения АТД путем комбинирования введенных ранее механизмов. Все перечисленные построения производятся в рамках динамического контроля типов.

5) В заключение рассматриваются некоторые статические свойства программ; рассматривается, как эти свойства отображаются в тексте программ и как в этих случаях путем трансформации текста программы можно переходить к статическому контролю типов.

4. Общие понятия языка ДАТД

Типы объектов. Будем считать, что в стандартном контексте описаны операции над объектами абстрактных стандартных типов, т. е. типы этих объектов недоступны, а известны только операции, которые осуществляют динамический контроль типов. Для построения новых объектов вводятся средства определения новых типов составных объектов и процедуры, как средство описания операций над объектами определяемых типов.

Стандартными типами являются скалярные типы с традиционным набором операций, один тип имени с операциями присваивания и явного разыменования, тип процедуры с операцией вызова, тип *тип* с предикатом *провтип* для проверки типа объекта. Вообще, все конструкции языка, операндом которых является тип, можно считать операциями над типами. Для стандартных типов в стандартном контексте введены обозначения, например, **имя**, **проц**. Они обозначают не сами типы (которые скрыты), а некоторые объекты, отличимые от всех других и поставленные в соответствие стандартным типам. В стандартных операциях, таких как *провтип*, это соответствие известно, так что, например, *провтип* (*x*, **проц**) проверяет, не является ли *x* объектом типа процедура. Этот прием необязателен, а используется он для того, чтобы сократить номенклатуру операций стандартного контекста.

Для скалярных объектов есть стандартные изображения. Для имен, процедур, составных объектов и новых типов вводятся генераторы. В качестве генератора процедурных объектов и типов используются конструкции, по традиции называемые соответственно изображением процедуры и изображением типа. Последняя используется для создания новых типов составных объектов. Для генерации имен и составных объектов используется общая конструкция «генератор».

Описание. Базовая форма описания и его семантика схожа с описанием тождества из Алгола-68. Этот простейший вид описания позволяет связать с некоторым объектом символьное обозначение (идентификатор), с помощью которого данный объект можно далее упоминать в программе. Принципиальное отличие от Алгола-68 состоит в том, что в описании не указывается тип объекта. Описание имеет следующий вид:

об *x* = *e*

где *x* — некоторый идентификатор, а *e* — ⟨выражение⟩.

В результате выполнения описания идентификатор начинает обозначать объект, вырабатываемый выражением. Поскольку конструируется язык класса Д, тип объекта в базовом описании статически не задается. Тип свойствен обозначаемому объекту и выра-

жение может, вообще говоря, при различных исполнениях описания вырабатывать объекты разного типа.

Описание типа имеет вид:

тип *m* = ⟨изображение Типа составного⟩

где *m* — идентификатор типа. В результате выполнения такого описания идентификатор начинает обозначать объект типа *тип*, вырабатываемый правой частью описания. Причина, по которой кроме описания тождества вводится особое описание типа, будет рассмотрена ниже.

Время жизни. В генераторе можно задать время жизни объекта. Для этого в генераторе определяется значение атрибута *локал*: *локал* : *e*. Значением выражения *e* может быть: (1) **истина**, тогда объект является локальным, (2) **ложь**, тогда объект считается глобальным, т. е. такие объекты могут уничтожаться с помощью техники сборки мусора, и (3) значением *e* может являться другой составной объект, тогда созданный объект прикрепляется к первому.

Локализация введена для того, чтобы отразить часто встречающийся случай, когда объект существует не дольше, чем активация процедуры или объемлющий составной объект. Поэтому по умолчанию создаваемый объект является локальным. Локальный объект прикрепляется к активации ближайшей текстуально объемлющей процедуры или, если для данного генератора ближайшим текстуально объемлющим является генератор другого составного объекта, то первый объект прикрепляется ко второму. Прикрепление к активации процедуры (к составному объекту) означает, что прикрепленный объект существует столько времени, сколько выполняется процедура (существует составной объект). Для ряда программ, разумеется, наиболее простой стратегией является ориентация на технику сборки мусора. С другой стороны, надо учесть, что есть системы, где нежелательны происходящие в непредсказуемые моменты времени прерывания работы программы для сборки мусора. Поэтому в общем случае дана возможность управления временем жизни объектов.

Квалификация. Чтобы реализовать принцип контекстной известности типа, нужно ввести контроль того, что декомпозиция объекта производится в контексте, где известен его тип. С этой целью вводится конструкция «вскрытия» объекта, имеющая вид $e\#m$, где *e* — выражение, а *m* — тип. Эту конструкцию будем называть «квалификация» *). При ее выполнении проверяется, что тип значения *e* есть *m*. Если проверка прошла успешно и квалификация вхо-

*) Этот термин взят из Симулы-67. Отличие от Симулы-67 состоит в том, что производится динамический контроль типа.

дит как составная часть в конструкции, где производится выбор элемента, то объект вскрывается и становится возможным доступ к его элементам. В других позициях тождественно выдается значение *e*. При несовпадении типа возникает ошибочная ситуация.

Основной смысл этой конструкции состоит в том, что контроль типа, включающий контроль известности типа, и декомпозиция объединяются в неделимое действие. Если в языке есть два отдельных примитива: предикат проверки типа и выбор элемента без контроля типа, то нельзя быть уверенным, например, в том, что между проверкой типа значения переменной и выбором элемента значения этой же переменной не произошло присваивание переменной объекта другого типа.

Контекстная цепочка. Чтобы ослабить связь между структурной и контекстной подчиненностью, в языке дана возможность явным образом указывать глобальный контекст, в котором происходит исполнение конструкций. Контекст вводится как цепочка (или список) элементов, которая называется контекстной цепочкой. Элементом цепочки, или секцией контекста, является структура. Контекст идентификаторов, образуемый цепочкой, составлен из идентификаторов элементов структур. Возможность конфликта идентификаторов для данного подхода не играет существенной роли. Считается, что доступны все элементы структур, входящих в цепочку контекста. Если же две структуры имеют одинаковые идентификаторы, то конфликт можно разрешить, например, путем введения средств именования элементов цепочки и обозначения конфликтующих элементов их идентификаторами, снабженными соответствующими префиксами.

Изображение цепочки называется контекстной приставкой и помещается перед конструкцией, для которой явно задается контекст. Приставка имеет вид:

контекст огр *em*1, *em*2, . . . **для**

где *em*1, *em*2,. . .— генератор структуры, или квалификация. В первом случае создается новая структура и включается в цепочку, во втором случае вскрывается существующая структура и включается в цепочку. Такой выбор конструкций гарантирует контекстную известность типа структуры, включаемой в цепочку. В результате исполнения конструкции создается контекстная цепочка, составленная из заданных структур. Если не указан символ **огр**, то в начало цепочки добавляется цепочка текущего контекста (т. е. контекста, действующего в конструкции, текстуально охватывающей место, где встречено изображение контекстной цепочки). Чтобы дать средство, обеспечивающее бóльшую степень независимости контекстных связей от структурных, введен символ **огр**. Он означает, что в цепочку не добавляется контекст охватывающей конструк-

ции. Это позволяет перечислить все необходимые структуры и ограничить только этим набором глобальный контекст конструкции, которой предшествует контекстная приставка.

5. Классы типов

Перейдем к анализу отдельных типов. На рис. 1 представлена графическая иллюстрация структуры объектов.

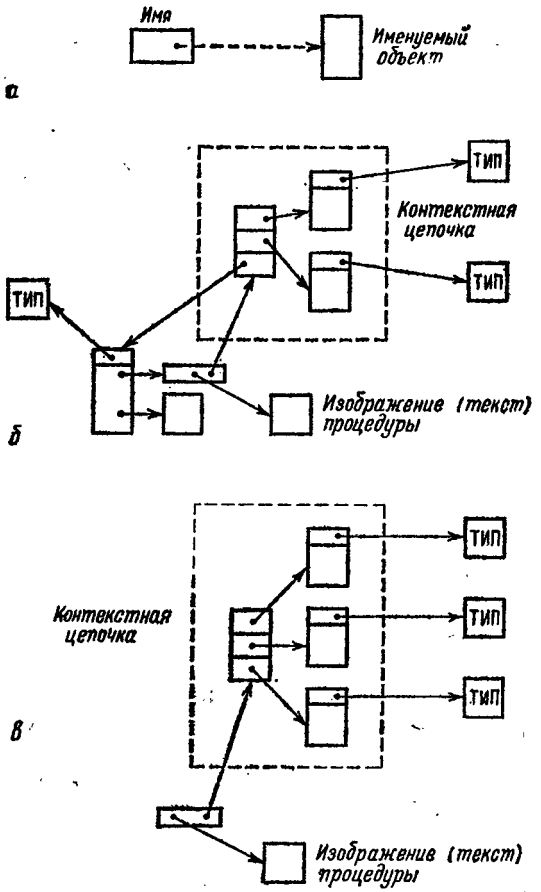


Рис. 1. Основные объекты языка ДАТД: *а* — имя; *б* — структура; *в* — процедурный объект

5.1. Имя. Изображение, соответствующее типу имени, имеет вид **имя**. Например, в результате выполнения **об** *x* **= ген имя** идентификатор *x* начинает обозначать созданное имя. К имени можно применять известные операции:

— присваивание, имеющее вид $x:=e$. На рис. 1, *а* связь имени с именуемым значением обозначена пунктирной стрелкой. В результате присваивания эта стрелка перенаправляется на присваиваемый объект;

— разыменование, имеющее вид x @. В языке с полной динамикой типов эта конструкция необходима из-за того, что тип объекта, обозначаемого идентификатором, неизвестен и поэтому нельзя произвести традиционное неявное разыменование имен. Ниже будут введены элементы статики, которые позволят преобразовать эту запись к обычному виду.

5.2. Структура. Изображение типа структуры имеет вид: **структ** (⟨список идентификаторов⟩). Структура представляет собой совокупность элементов, являющихся объектами (рис. 1, *б*). Каждый элемент снабжен идентификатором, который указан в изображении типа и предназначен для обозначения соответствующего составляющего объекта. Например, описание типа структуры может иметь вид:

тип m = **структ** (**об** *а*, *в*).

Список идентификаторов может быть пустым. Это используется в тех случаях, когда не важна логическая структура объекта, а просто нужен тип, отличный от всех других, и объекты, отличные от всех других.

Структура типа m создается с помощью генератора, который в общем случае имеет вид:

⟨контекстная приставка⟩ **ген** m (⟨описания⟩) ⟨время жизни⟩

Например:

контекст $e_1 \# m_1$, $e_2 \# m_2$ **для ген** m (**об** $a = ea$, $в = ев$)

Описания приводятся для всех идентификаторов компонентов, заданных в изображении типа. В общем случае после ген может идти не только идентификатор типа, но и непосредственное изображение типа, использование которого в процессе исполнения генератора приводит к созданию нового анонимного типа. Такое обобщение ничему не противоречит, но, как будет показано в дальнейшем, удобно с практической точки зрения.

Формально генерация объекта происходит следующим образом. Сначала из контекстной приставки создается глобальная контекстная цепочка, затем создается объект заданного типа с «неопределенными» элементами, который добавляется в контекстную цепочку. После этого в данном контексте выполняются доопределения элементов вновь созданного объекта. Для этого исполняются правые части описаний генератора, и объекты, полученные в результате исполнения, становятся соответствующими элементами вновь

созданного объекта; типы объектов-элементов могут быть любыми. Если, например, *ea* представляет собой изображение процедуры без собственной контекстной приставки, то элемент *a*, являющийся объектом типа процедура, будет контекстно-подчинен созданной структуре, см. рис. 1, *б*.

Введем одно естественное сокращение. Вместо **ген структ (об** *a*, *в*, . . .) **(об** *a*=*ea*, *в*=*ев*, . . .) можно писать **ген (об** *a*=*ea*, *в*=*ев*, . . .). Исходная запись удобнее в тех случаях, когда в доопределяющей части есть два описания, каждое из которых использует идентификатор, введенный в другом описании. Отсюда видно происхождение так называемых предварительных описаний в ЯВУ, где описание идентификатора должно предшествовать его использованию; в исходной записи роль таких описаний выполняет список идентификаторов в изображении типа. Переходя к традиционной нотации, надо «совместить» эти два списка, допустив описания вида **об** *x* без правой части. Такое описание рассматривается в ДАТД как предварительное.

Конструкция выбора элемента структуры имеет вид $e\#m.a$, где *a* — идентификатор элемента структуры типа *m*, а *e* — ⟨выражение⟩, значением которого должен быть объект типа *m*. В результате выполнения конструкции выдается обозначенный элемент данного объекта (т. е. выдается объект, входящий в состав данного объекта). Как видим, конструкция для декомпозиции объекта устроена так, что контролируется известность типа объекта в данном контексте и правильность типа. С помощью этой конструкции в языке ДАТД реализуется принцип контекстной известности типов, так как оказывается, что доступ к элементам объекта можно получить лишь при условии «предъявления» типа объекта. Если тип неизвестен, то невозможен доступ к элементам. В отличие от этого конструкция вида *e.a* (или подобные ей в языках типа EL/1), где *e* — выражение статически неизвестного вида, дает доступ к элементам любого объекта, находящегося в текущем контексте.

Не противореча общим принципам, можно допустить, чтобы в конструкции выбора элемента на месте квалификации мог быть генератор объекта, так как он тоже указывает, что тип объекта известен в данном контексте.

5.3. Массив. Основные отличия средств обработки массивов связаны со способом обозначения элементов (элемент массива обозначается не идентификатором, а индексом), а также есть ряд соглашений, упрощающих программирование обработки массивов в случае, если массив является гомогенным составным объектом. Пусть, например, описание типа массива имеет вид:

тип *m* = **массив** [ ]

В результате выполнения генератора **ген** *т* [*к*] (*е*) создается составной объект, представляющий собой массив из *к* объектов, каждый из которых является результатом выражения *е*. В частности, если *е* — это генератор объекта типа имя (например **ген имя**), то создается традиционный одномерный массив имен (или массив переменных). Массив массивов также строится очевидным способом. Формально нет необходимости вводить ограничение, требующее, чтобы все элементы массива имели один и тот же тип. Например, можно для разных индексов (или диапазонов индексов) задавать объекты разных типов:

ген *т* (*к*) (1. . .*м*1 = *е*1, *м*2.. *к* = *е*)

Графическое изображение массива отличается от изображения структуры на рис. 1,*б* способом обозначения компонентов. Конструкция для выбора элемента массива («индексация») строится так же, как для выбора элемента структуры с учетом различия в обозначении элементов этих двух классов объектов. Она имеет вид *е*##*т* [*к*], где *т* — тип массива. В дальнейшем будем считать, что в стандартном окружении описан тип одномерного массива, обозначаемый *вект*, и процедура генерации массива переменных типа *вект*. Индексацию вида *е* [*к*] будем понимать как сокращенное обозначение для *е*##*вект* [*к*]. Это сокращение допустимо, поскольку тип *вект* глобально известен и, следовательно, если только массив такого типа доступен в контексте программы, то программа имеет доступ и к его элементам.

5.4. Процедура. Изображение процедуры без локальных описаний имеет вид:

⟨контекстная приставка⟩ **проц** (⟨список идентификаторов⟩) (⟨предложения⟩)

⟨список идентификаторов⟩ в скобках рассматривается как изображение типа структуры формальных параметров. Исполнение изображения состоит в том, что исполняется контекстная приставка и создается объект типа процедуры, формально представляющий собой совокупность из полученной контекстной цепочки и текста исходного изображения процедуры, рис. 1, *в*. Вызов процедуры имеет традиционный вид: *е* (e_1, e_2, . . .), где значением *е* должен быть процедурный объект, а значениями выражений e_1, e_2, . . . являются объекты — фактические параметры, типы которых могут быть любыми.

Семантика вызова трактуется, как обычно, за исключением того, что совокупность параметров в процессе выполнения предложений процедуры рассматривается как объект типа структуры. После вычисления фактических параметров создается объект типа структуры формальных параметров, прикрепляемый к активации

вызываемой процедуры. Его элементами (в соответствии с порядком следования в списках формальных и фактических) становятся фактические параметры. Далее берется контекстная цепочка из процедурного объекта, в нее добавляется созданная структура и в полученном контексте выполняются предложения из изображения процедуры.

Пустой список идентификаторов в изображении процедуры и пустой список выражений в вызове задают соответственно изображение и вызов процедуры без параметров. Круглые скобки в этих случаях остаются.

Теперь мы ввели все необходимые конструкции; перейдем к примерам, иллюстрирующим некоторые новые возможности этих конструкций.

5.5. Примеры процедур. Рассмотрим три частных случая изображения процедуры. Первый случай иллюстрирует тот факт, что локальный генератор в контекстной приставке можно использовать так, что получается процедура, эквивалентная процедуре с локальными описаниями. Во втором случае показано применение генератора в контекстной приставке, дающее процедуру, эквивалентную процедуре с собственными величинами в Алголе-60. Третий пример показывает, что время жизни процедурного объекта не обязательно ограничено временем выполнения текстуально охватывающей процедуры.

Пример 1. Процедура с локальными описаниями.

Эквивалентом традиционной процедуры с локальными описаниями является такая процедура:

⟨контекстная приставка⟩ **проц** (⟨список идентификаторов⟩)
(контекст ген (⟨описания⟩) **для**
проц ( ) (⟨предложения⟩) ( ))

Описания, заключенные в скобки, являются эквивалентом локальных описаний. Заметим, что структуры формальных параметров и локальных данных в соответствии с этим определением и определением семантики вызова создаются как объекты, которые по умолчанию имеют локальное время жизни. Это полностью соответствует традиционной семантике создания и уничтожения параметров и локальных, а также стековому механизму реализации процедур. Естественно ввести сокращенную и более традиционную запись для таких процедур:

⟨контекстная приставка⟩ **проц** (⟨список идентификаторов⟩)
(⟨описания⟩; ⟨предложения⟩)

Подчеркнем, что понимать такую запись будем в том смысле, который имеет исходное изображение. Декомпозиция изображения процедуры на контекстную приставку и изображения типов структур

формальных параметров и локальных данных понадобится в дальнейшем при построении структурной модели процедуры.

Пример 2. Процедура с собственными величинами.

Механизм собственных величин Алгола-60 программируется так:

```
контекст ген (описания собственных величин) для
проц (. . .) (. . .)
```

В результате выполнения генератора в контекстной приставке создается структура, которая вследствие ее анонимности доступна только данной процедуре. Она создается при исполнении изображения, т. е. перед первым вызовом данной процедуры, и будет сохранять информацию от одного вызова этой процедуры до другого.

Пример 3. Выдача процедуры в качестве результата выполнения другой процедуры.

Средства явного указания контекста процедуры позволяют задавать время жизни процедурного объекта вне связи с временем выполнения текстуально объемлющей процедуры, т. е. время жизни процедуры оказывается ограниченным только временем жизни объектов, включенных в цепочку ее глобального контекста. Рассмотрим изображение процедуры *а*:

```
проц (. . .) (. . .
              об в = контекст огр . . . для проц (. . .) (. . .);
              . . .; в)% выдача процедуры
```

Эта процедура запрограммирована так, что она выдает процедуру *в* в качестве результата исполнения. При этом контекстную приставку *в* можно организовать так, чтобы сохранялись все необходимые объекты, доступные внутри *а* (например, переданные через параметры), но имеющие время жизни большее, чем время выполнения процедуры *а*. Отметим, что структурная подчиненность влечет за собой определенную контекстную подчиненность в том смысле, что результирующий контекст может быть только сужен, но не расширен. Это происходит из-за того, что контекстная приставка выполняется в контексте охватывающей процедуры.

6. Пакет процедур обработки АТД

Перейдем к примеру программирования пакета процедур обработки объектов АТД. Для этого введем описание типа пакета как структуры, компонентами которой являются доступные пользователю процедуры пакета. Изображение типа структуры можно называть внешними спецификациями пакета. Затем опишем процедуру генерации экземпляра такой структуры. Ее будем называть внешней частью пакета, в отличие

от скрытой части пакета, которая тоже создается при генерации. Отметим еще раз, что мы не вводим каких-либо специальных средств для организации пакетов, а пользуемся общими понятиями процедуры и структуры. Одно из частных следствий этого подхода состоит в том, что можно описать несколько процедур генерации. С практической точки зрения они будут задавать различные способы реализации внешних спецификаций пакета.

Пусть тип пакета имеет вид:

```
тип пак = структ (об созд, оп)
```

где *созд* будет обозначать процедуру, создающую объекты АТД и выдающую их в качестве значения, а *оп* — некоторую операцию обработки объекта АТД. Процедура генерации пакета может иметь вид:

```
об генпак = проц (вж) % вж — параметр,
                        % определяющий время жизни
                        % экземпляра пакета
( контекст огр . . . для ген
 (тип т = структ (. . .); % скрытый тип
   об скрперем = ген имя, % скрытая переменная
      скрпроц,     % скрытая процедура
      в = ген пак (
      об созд = проц ( ) (ген т (. . .) (локал : в)),
      оп = проц (х) (. . .х#т. . .скрперем. . .скрпроц. . .)),
      скрпроц = контекст в#пак для проц (. . .) (. . .)
 ) (локал: вж). в% выборка элемента в
) % конец генпак
```

Здесь тело процедуры представляет собой конструкцию выбора элемента структуры, причем перед символом «.» использован генератор структуры, в котором изображение типа совмещено с описаниями компонентов.

Основные особенности этой процедуры состоят в следующем:

1. В результате вызова пользователю выдается не весь созданный объект, а только его часть *в*, состоящая из доступных пользователю процедур. Полная структура экземпляра пакета, создаваемого процедурой генерации, представлена на рис. 2. Приведем пример программирования процедуры, использующей экземпляр пакета:

```
контекст генпак (лок)#пак для
проц (. . .)
(об х = созд ( );
        . . .оп (х) . . .)
```

В контекстной приставке создается экземпляр пакета, а внутри этой процедуры используются операции данного пакета.

2. Тип *т* скрыт от пользователя пакета. Так, например, объект *х* можно только передать в процедуру *оп*. Внутри процедуры *оп* тип *т* известен, и поэтому данная процедура может обрабатывать элементы объекта.

3. Параметром процедуры генерации пакета является объект, к которому прикрепляется экземпляр скрытой части пакета. К последнему по умолчанию прикрепляется экземпляр внешней части

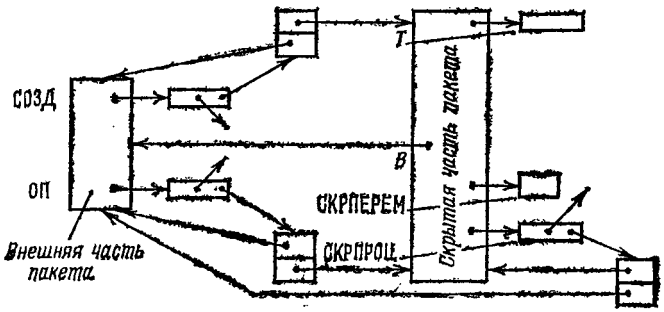


Рис. 2. Структура пакета после генерации. (Изображения процедур на рисунке не обозначены)

пакета. Таким способом можно программировать управление временем жизни пакета. Кроме того, процедура *созд* запрограммирована так, что создаваемые ею объекты имеют такое же время жизни, как и весь пакет. Несложное изменение этой процедуры позволяет задавать управление временем жизни, индивидуальное для каждого объекта:

созд = **проц** (*лок*) (**ген** *т* (. . .) (*локал* : *лок*))

Если программист предполагает воспользоваться сборкой мусора, то он может задавать глобальное время жизни для пакета и создаваемых им объектов. В этом случае вызов процедур генерации программируется так: *генпак* (**ложь**) и *созд* (**ложь**). Рассмотренный вариант структуры пакета, разумеется, не является единственно возможным. Например, процедура генерации может быть запрограммирована так, что объект *в* не является элементом основного объекта.

Практическое ограничение рассмотренного подхода состоит в том, что описания типа пакета и его реализаций должны быть сосредоточены в одной программе с их использованием. Если, как развитие механизма управляемого контекста, ввести в язык возможность

создавать и именовать объекты архива, являющиеся типами и процедурами, то это ограничением будет снято. Вариант такого подхода рассматривается в § 8.

7. Статические свойства объектов

Рассмотрим статические свойства объектов и статические соотношения между объектами, т. е. свойства и соотношения, сохраняемые в процессе использования объектов в программе. Наиболее полно этот вопрос отображен в языках спецификаций. Здесь рассматриваются те статические свойства, которые задаются в процедурных языках программирования с помощью статики типов.

Для данного анализа этот вопрос важен потому, что переход от динамического контроля к статическому основан на учете статических свойств объектов. Подчеркнем, что ниже излагается подход к разработке языка, ядро которого основано на динамике типов, и вместе с тем есть средства, позволяющие управлять степенью динамизма контроля типов, приспосабливая к природе конкретных алгоритмов и структур данных. Например, возможны структуры, одна часть элементов которых имеет статически известный тип, а тип других элементов зависит от динамики выполнения программы. Аналогично могут быть устроены параметры процедуры. Для некоторого имени тип именуемого объекта может быть зафиксирован, но тип элементов этого объекта статически неизвестен и т. д.

7.1. Фиксированный контекст. Во-первых, отметим, что рассмотренное выше свойство процедурного объекта, состоящее в том, что связь этого объекта с глобальным контекстом фиксируется в момент создания объекта и сохраняется в процессе его использования, можно считать статическим свойством этого объекта, на котором основан механизм распределенного контекста.

Относительно элементов контекстной цепочки надо сказать следующее. Для задания элемента цепочки была использована квалификация вида $x\#\#m$. Здесь она применяется в качестве общего приема, обеспечивающего доступ к элементам объекта в том случае, если известен его тип. Что же касается той особенности, что объект x типа m имеет статически фиксированный набор элементов, этот момент с защитой не связан. Это статическое свойство объектов типа m, имеющее отношение скорее к реализации, чем к существу модели языка. В языках компиляционного типа, как и в языке ДАТД, оно используется для того, чтобы статически распределять память внутри объекта и соответственно выбор элемента объекта компилировать в код, содержащий статическое смещение относительно базового адреса.

Этим же объясняется то, что в конструкции «квалификация» тип задается идентификатором типа, а не выражением, вырабаты-

вающим объект типа *тип*, и также то, что введено специальное описание типа, вместо того чтобы иметь в языке только одно описание тождества, используемое, в частности, для описания типов. Если снять эти ограничения, оставив механизм распределенного контекста, изменится реализация выбора элемента, но контекстная защита и механизм АТД сохраняются. Например, это видно из дальнейшего изложения, когда будут введены средства архивного именования. Ввиду того, что архивные символьные имена создаются в ходе диалога человека с вычислительной системой, этот процесс имеет существенно динамический характер, и здесь практически нецелесообразно ограничиться компиляционным вариантом. Однако независимо от этого можно обеспечить контекстную защиту.

7.2. Фиксированная процедура генерации. Если объекты некоторого типа *т* всегда создаются с помощью одной и той же процедуры генерации вида

проц (*вж*) (**ген** *т* (⟨описания⟩) (*локал* : *вж*)),

то изображение данного типа можно разрешить представлять в сокращенном виде: **структ** (⟨описания⟩). При генерации объекта такого типа происходят те же действия, что и при вызове изображенной выше процедуры. Формальный параметр *вж* становится неявным параметром типа. Фактическое значение этого параметра задается с помощью атрибута *локал* в генераторе объекта данного типа.

7.3. Фиксированный тип элемента. Т-выражением будем называть выражение, тип значения которого статически известен. В ядре языка ДАТД уже есть два вида Т-выражений — квалификация $e\#\#m$ и генератор **ген** *т*. Для общности можно разрешить использовать на месте идентификатора типа обозначения для эквивалентов (см. § 3) стандартных типов: **имя**, **проц** и т. д. (в конструкции выбора использование этих обозначений не имеет смысла, и там это приводит к ошибке). Далее рассмотрим статическое свойство составного объекта типа *т*, состоящее в том, что его элементом *х* всегда является объект типа *тх*. Потребуем, чтобы в программе это свойство выражалось в том, что описание элемента имеет вид: **об** $x = etx$, где *etx* — Т-выражение типа *тх*.

Если это свойство имеет место, то в изображении типа *т* оно указывается в списке идентификаторов так: $x\#\#mx$. Аналогично изменим вид левой части описания элемента на **об** $x\#\#mx$. Везде в программе использующее вхождение *х* будем считать эквивалентным конструкции квалификация $x\#\#mx$, т. е. отнесем *х* к Т-выражениям типа *тх*. Тем самым, в сущности, введен статический контроль типа элемента *х*.

Теперь, следуя языковым традициям, можно ввести учет контекстной позиции, в которой происходит однократное неявное разыменование. Если из описания идентификатора следует, что он обоз-

начает объект типа имя, то вместо $x@$ надо писать просто x. Отмена разыменования в этой позиции обозначается $@x$.

Тип элемента массива. Поскольку индекс элемента массива в общем случае вычисляется в программе динамически, фиксация типа только какого-либо отдельного элемента не дает существенного результата в части статического контроля. Поэтому для массива будем рассматривать случай, когда все его элементы являются объектами одного типа. Изображение типа такого массива имеет вид:

массив [ ]#*m*

7.4. Описание, совмещенное с генерацией. Рассмотрим частный случай только что введенного описания: об $x\#m$ = **ген** m. Здесь во время выполнения описания происходит генерация объекта. Такое описание, совмещенное с генерацией, характерно для языков со статическим контролем типа. Оно будет кратко обозначаться: **об** $x : m$, что соответствует традиции. Тогда может быть дано, например, такое описание типа структуры, состоящей из двух переменных:

тип *mc* = **структ** (**об** *а* : **имя**, *в* : **имя**)

Описание объектов такого типа имеет вид:

об *с* : *mc*, *д* : *mc*

Элементы *с* и *д* обозначаются в программе, как обычно, *с.а*, *д.а* и т. д.

Введем еще одно соглашение по поводу генератора имен. Неограниченное использование этой конструкции создает неудобство, поскольку тут появляется (по-видимому, излишняя) возможность неявного разыменования. Это снижает степень выразительности программ и создает определенные реализационные проблемы. В связи с этим допустим использование генератора имени только в правой части описания и введем соответствующее сокращение: вместо **об** *x* : **имя** будем писать **перем** *x*. Тем самым непосредственным образом введено описание переменной. После того, как исключены все случаи явного указания типа имени, описание вида:

об *x*1; **об** *x*2 = *e*; **об** *x*3#*m* = *e*; **об** *x*4 : *m*

будем записывать в форме описания констант:

конст *x*1; **конст** *x*2 = *e*; **конст** *x*3#*m* = *e*; **конст** *x*4 : *m*.

Аналогичным образом вводятся соглашения относительно типов. Опуская промежуточные построения, дадим форму окончательной записи:

— тип массива переменных : **массив** [ ] **перем**

— тип массива констант : **массив** [ ] **конст**

— тип массива константт типа *тх* : **массив** [ ] **конст**#*тх*

— тип массива констант типа *тх*, значение которых создается при генерации массива: **массив** [ ] **конст** : *тх*.

Если массив типа **тип** *тм* = **массив** [ ]. . . всегда создается генератором вида **ген** *тм* (*к*), то тип массива можно сокращенно обозначать **массив** [*к*]. . .

Возможность задавать изображение типа непосредственно в описании константы вводится следующим образом. Если после замены всех вхождений *х*#*тх* на *х* оказалось, что идентификатор *тх* используется в программе только в описании *х*, то можно подставить правую часть описания *тх* в описание *х*, т. е. заменить это описание на: **конст** *х* : ⟨изображение типа⟩, одновременно убрав описание *тх*, т. е. *тх* стал так называемым «анонимным» типом. В результате приходим к описаниям, эквивалентным традиционным описаниям переменных, структур и массивов (с тем отличием, что пока нет возможности фиксировать тип значения переменной).

Приведем несколько примеров:

— описание переменных

перем *х*, *у*

с обозначением *х* :=*у* для присваивания переменной *х* значения переменной *у*;

— описание структуры с элементами-переменными

конст *х* : **структ** (**перем** *а*, *в*)

с обозначением компонентов *х.а* и *х.в*.;

— описание массива переменных

конст *х* : **массив** [. . .] **перем**

с обозначением компонентов *х* [. . .];

— описание массива структур

конст *х* : **массив** [. . .] **конст** : **структ** (**перем** *а*, *в*)

с обозначением компонентов *х* [. . .] .*а* и *х* [. . .] .*в*.

Особо подчеркнем, что не вводятся два класса объектов: класс статически контролируемых объектов и класс динамически контролируемых объектов. Например, в контексте описаний *тс* и *с* возможно следующее их использование:

перем *м*;
. . . *м* : =*с*; . . . *м*#*тс.а* . . .
конст *к*#*тс* = *м*#*тс*; . . . *к.а* . . .

Здесь объект *с*, полученный в результате выполнения «статического» описания, становится значением переменной *м* динамиче-

ского типа. Обратиться к элементу объекта, являющегося значением такой переменной, можно только с помощью квалификации, которая осуществляет динамический контроль типа. Описание *к* можно понимать как описание константы статического типа. Если при инициализации этой константы используется значение переменной *м*, то тип этого значения должен быть проконтролирован динамически. Последующий доступ к элементам *к* производится уже без динамического контроля типа.

Конструкция «квалификация», как видим, используется с несколькими целями: (а) для контроля прав доступа к объекту; (б) для динамического контроля типа; (в) с точки зрения реализации эта конструкция в сочетании с описанием типа позволяет компилировать выбор компонента в статическое смещение относительно базы объекта вместо поиска элемента по символьному имени.

7.5. Фиксированный тип переменной и параметров. Аналогичным образом, рассматривая случай, когда некоторой переменной всегда присваивается значение Т-выражения одного и того же типа *mx*, можно прийти к описанию переменной статического типа **перем** *x*##*mx*. Например, описание

конст *а* : **массив** [10] **перем**##*цел*

семантически эквивалентно следующему описанию в языке Паскаль:

var a : *array* [10] *of integer*,

поскольку в обоих случаях массив состоит из переменных целого типа и создается в момент выполнения описания.

Схожий прием приводит к процедурам со статическим контролем типа параметров и/или типа результата. Например, описание константы, значением которой является процедура-функция со статическим контролем параметров и результата, может иметь вид

конст *р*##**проц** ТП##*т*,

где ТП — тип структуры формальных параметров, а *т* — тип результата. Причем ТП может быть устроен так, что только для некоторых параметров задан статический контроль; число параметров при таком описании всегда контролируется статически из-за того, что статически фиксировано число элементов структуры.

С точки зрения программирования существенно то, что в целом изложенный подход позволяет выбирать необходимую для данной программы степень динамизма (или статичности) контроля типов. Вместе с тем отметим, что с введением статики компактный поначалу язык начинает загромождаться дополнительными конструкциями.

7.6. Скрытые типы. В примере § 6 тип *т* был полностью скрыт от пользователя пакета. Тем самым было показано, что в базисном языке для введения скрытых типов нет необходимости иметь какие-

либо специальные конструкции. Тем не менее, допустив возможность статического контроля типа параметров, естественно ввести соответствующее сокращение записи для скрытых типов. Если параметр и/или результат некоторой процедуры пакета всегда имеет тип *т*, причем этот тип является скрытым типом данного пакета, то в изображении типа пакета можно дать предварительное описание типа *т*, т. е. описание типа без правой части. В этом случае обеспечивается статический контроль, и в процедуре *оп* квалификация *х*#*т*, выполняющая динамический контроль типа, становится ненужной:

тип *пак* = **структ** (**тип** *т*; **конст** *созд*#**проц** (. . .)#*т*,
оп#**проц** (**конст** *х*#*т*)#. . .)

Полное описание указывается в процедуре генерации экземпляра пакета. При этом сохраняется возможность иметь несколько процедур генерации экземпляра пакета, в которых тип *т* может быть описан по-разному. Таким образом, отличие от других языков в этой части состоит в том, что:

— это средство не является базовым. В общем случае, когда не соблюдается указанное выше статическое соотношение, можно пользоваться базовым механизмом динамического контроля типов;

— в спецификациях пакета не указывается полное описание скрытого типа. Это позволяет изменять структуру объектов скрытого типа или иметь несколько разных вариантов этой структуры, не меняя спецификации пакета и не перетранслируя программы пользователей.

8. Внешние объекты

В этом параграфе будут введены элементы языка ДАТД, связанные с обработкой архивных или внешних объектов. Под термином внешние объекты понимаются объекты, время жизни которых не ограничивается временем выполнения программ. Такой объект может существовать до начала выполнения программы, которая его использует, и продолжать существовать по окончании выполнения. Применяемые в БПК способы обработки внешних объектов включают вызов программ из архива, компиляцию программ, обработку файлов, создание новых внешних объектов и уничтожение старых, другие архивные операции, взаимодействие с пользователем путем записи сообщений в терминальный файл и считывание ответных сообщений. Все это определяет достаточно существенные требования к языку программирования.

При разработке средств управления внешними объектами можно отдельно рассматривать два аспекта: (1) разработка средств именования внешних объектов и механизмов, определяющих время

их жизни, и (2) разработка средств структурирования внешних объектов (описание типов объектов и их взаимосвязи).

8.1. Именование и время жизни. Этот аспект играет достаточно самостоятельную роль и зависит от второго в основном в том отношении, что с введением новых средств структурирования объектов расширяется номенклатура конструкций для обозначения элементов, тогда как базовый механизм распределенного контекста сохраняется. То же самое можно сказать о методе определения времени жизни и других общезначимых механизмах: передача параметров, присваивание и пр., см. п. 2.4. В данном случае реализация принципа универсальной применимости базовых средств обработки объектов должна состоять в том, что эти механизмы «работают» и для внешних объектов. Как показано в [1], в традиционных ЯВУ это правило не соблюдается.

Форма, в которой в языке представлены средства именования и управления временем жизни, влияет на следующие свойства языка: — п о л н о т а языка в отношении описания всего цикла обработки объекта (создание объекта — последующее обращение к объекту по имени — уничтожение объекта). Например, если некоторые этапы надо описывать на другом языке, то можно заключить, что первый язык неполон;

— к о н ц е п т у а л ь н а я о д н о р о д н о с т ь или к о н ц е п т у а л ь н о е е д и н с т в о языка, зависящие в рассматриваемом случае от того, насколько средства обработки внешних объектов соответствуют принципам обработки других объектов. Например, неоднородность может выражаться в том, что при работе с внешними объектами не осуществляется контроль типов.

Эти свойства в свою очередь влияют на уровень языка в части обработки внешних объектов. За счет однородности преимущества ЯВУ, связанные с контекстной защитой и контролем типов, можно распространить на программирование обработки внешних объектов. За счет полноты можно обойтись одним данным языком при описании различных этапов обработки внешних объектов.

Ниже в этом параграфе будет рассмотрен подход к разработке средств управления внешними объектами для процедурного ЯВУ, основанный на том, что базовые механизмы ЯВУ отображаются на класс внешних объектов.

8.2. Базы данных. Возможности описания структуры внешних объектов и отношений между ними в значительной мере определяют уровень языковых средств обработки этих объектов. Применяемая для этого техника баз данных (БД) значительно повышает уровень работы пользователя с информацией. Она же может упростить разработку системных программ. Наконец, БД играют центральную роль в перспективных системах программирования как адекватная основа для организации хранения модели предметной области и

данных из различных предметных областей. Вместе с тем можно сказать, что ряд механизмов ЯВУ продолжает сохранять свое фундаментальное значение. В первую очередь это касается механизма распределенного контекста, упрощающего решение проблемы защиты от несанкционированного доступа к БД.

ЯВУ с некоторым набором встроенных типов внешних объектов и механизмом контекстного именования внешних объектов можно рассматривать как инструментальное средство для разработки системы управления БД (СУБД), осуществляющей проекцию объектов БД на объекты стандартных типов. Рассматривая СУБД как пример БПК, функционирование которого в основном представляет собой процесс обработки внешних объектов, можно сделать вывод о том, что полнота и однородность языка играют здесь важную роль, поскольку позволяют всю СУБД программировать на данном языке. В настоящее время уже имеется положительный опыт применения Эль-76 для разработки конкретных СУБД и в первую очередь для реализации СУБД МВК Эльбрус [15].

8.3. Внешний контекст. Перейдем к рассмотрению элементов ДАТД, связанных с управлением внешними объектами. Введем понятие внешнего контекста процедуры, логически эквивалентное понятию контекста процедуры. В основе понятия внешнего контекста лежит следующая языковая модель работы пользователя в системе (рис. 3) [24].

Активность пользователя рассматривается как процесс выполнения особой процедуры, которую можно назвать процедурой активности пользователя. Процесс ее выполнения начинается в момент регистрации пользователя в системе. На каждом сеансе выполняется несколько очередных операторов, введенных в пакетном или диалоговом режиме. В результате их выполнения могут создаваться новые объекты, использоваться или ликвидироваться созданные на предыдущих сеансах. Можно сказать, что время выполнения этой процедуры совпадает со временем, в течение которого пользователь зарегистрирован в системе, а ее объекты обработки распределены в архиве пользователя.

Эта модель привносит в язык одну новую черту. Необычность процедуры активности пользователя, связанная как раз с тем, что она моделирует вычислительную деятельность человека, состоит в том, что в ходе выполнения в контекст могут вводиться произвольные, заранее неизвестные системе идентификаторы и удаляться существующие. Обработку таких имен целесообразно производить не компиляционно, а интерпретационно, путем ассоциативного поиска символьного имени в контексте обозначений. Чтобы учесть эту особенность, и вводится упомянутое выше понятие внешнего контекста процедуры или контекста внешних имен, поиск в котором производится по имени в процессе выполнения.

Этот контекст можно наращивать или сокращать. Можно считать, что секцией внешнего контекста является структура стандартного абстрактного типа «справочник». Отличительной особенностью этой структуры является то, что к ней можно добавлять новые элементы и удалять существующие. Элемент справочника снабжен символьным обозначением и является эквивалентом объекта типа имя в том смысле, что подобное имя может именовать (ссылаться на) объект, находящийся во внешнем контексте. Такие имена

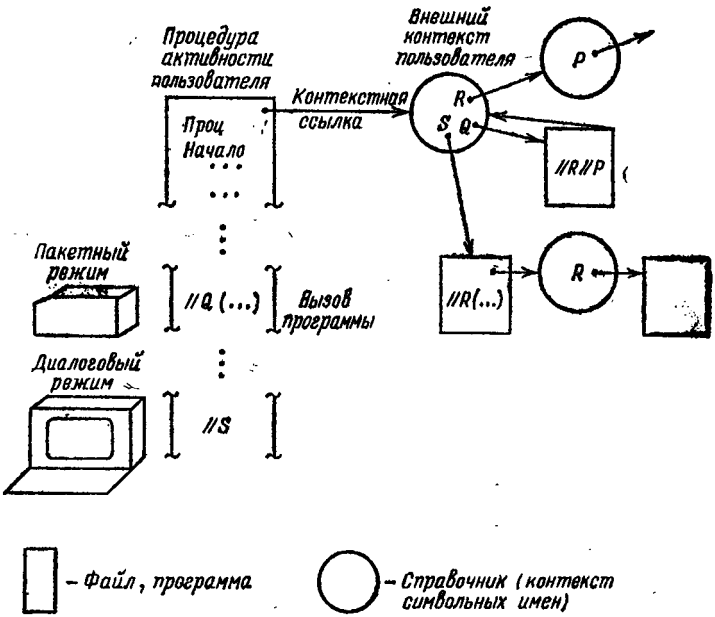


Рис. 3. Модель работы пользователя

назовем указателями внешних объектов или файловыми ссылками,— это так называемые внешние объекты. Особенность такого объекта состоит в том, что он может существовать дольше, чем выполняются использующие программы.

Реализационные соображения позволяют использовать для внешних объектов наиболее гибкую стратегию определения времени жизни, а именно концепцию удерживания. Если считать, что накладные расходы на обработку внешних объектов велики (по сравнению, например, с обработкой массивов), то реализация механизма удерживания, например, с помощью счетчика ссылок на объект будет составлять малую долю от общих расходов. Поэтому будем считать, что объект продолжает существовать (удерживается), пока

не закончилось выполнение процедуры активности, и есть некоторый путь доступа из этой процедуры к объекту. Когда не останется такого пути доступа, объект будет автоматически уничтожен.

После завершения процедуры активности, т. е. после исключения пользователя из списка пользователей, этот процедурный объект будет уничтожен. Если при этом оказывается, что не осталось ссылок на секцию его внешнего контекста, то она будет уничтожена, что приведет к уничтожению объектов архива пользователя, кроме тех, к которым остался путь доступа из процедур активности других пользователей *).

Кроме архивов пользователей, в системе обычно существует глобальный архив. Объекты этого архива удерживаются за счет того, что выполняется постоянно действующая (по определению) процедура активности системы (или процедура активности особого пользователя — оператора системы). Во внешнем контексте этой процедуры, кроме системных объектов, доступных только данной процедуре, находится справочник доступных пользователю объектов глобального архива. С формальной точки зрения эта процедура при регистрации нового пользователя создает его внешний контекст, состоящий из одной стандартной ссылки на справочник глобального архива, и запускает в этом контексте процедуру активности данного пользователя.

Эта гипотетическая картина соответствует тому, что происходит реально в вычислительной системе. Фактически в работе пользователя есть перерывы между сеансами. Но за счет того, что ссылка на внешний контекст пользователя помещается еще и в системный архив (в список пользователей), внешние объекты пользователя удерживаются во время перерыва между сеансами. Тем самым моделируется постоянное действие процедуры активности пользователя.

Учитывая эту модель работы пользователя, а также некоторые реализационные соображения, в системе Эльбрус введена следующая иерархия макрообъектов или макропроцессов и соответствующие ей ограничения на время жизни объектов:

— пользователь. Пока существует этот объект, могут существовать внешние объекты, к которым есть путь доступа из внешнего контекста пользователя;

— выполняемая на некотором сеансе последовательность операторов процедуры активности пользователя. С этой последовательностью связан объект, называемый з а д а ч а. Происхождение этого объекта имеет в основном практический характер. Задача — это потенциальный ресурс математической памяти, который можно ис-

---

*) Предполагается, что есть встроенные операции организации «почты», позволяющие одному пользователю передавать свои объекты другому.

пользовать во время сеанса *). В этой памяти создаются оперативные объекты, время жизни которых ограничено временем существования задачи, т. е. временем сеанса;

— процесс. В рамках задачи или процесса может быть вызвано несколько параллельно выполняемых процедур, каждая из которых образует процесс;

— вызов процедуры имеет традиционный смысл. Объекты, локализованные в процедуре, уничтожаются по окончании ее выполнения.

8.4. Типы внешних объектов. К числу внешних объектов отнесем:

1. Объекты стандартного абстрактного типа «справочник» с операциями создания /уничтожения объекта, создания/ уничтожения элемента с заданным обозначением, присваивания элементу и разыменования.

2. Объекты стандартного абстрактного типа «простой неструктурированный файл» с операциями создания /уничтожения файла, чтения/ записи информации.

3. Из числа объектов, введенных в § 3, к внешним можно отнести такие объекты, время жизни которых не связано с временем жизни оперативных объектов и которые, следовательно, могут существовать после завершения задачи. Таковыми являются статические объекты. Статические объекты — это объекты, свойства которых известны при компиляции и не изменяются в процессе их обработки. Можно считать, что такие объекты или создаются в результате компиляции, или априорно существуют в стандартном окружении. К статическим объектам относятся объекты встроенных скалярных типов, составные объекты, элементами которых являются статические объекты, процедурные объекты и типы, контекстно подчиненные статическим структурам. Тип статического внешнего объекта или тип параметров должен быть внешним объектом.

Введенные в п. 3 внешние объекты можно назвать программами. Обычно внешний процедурный объект контекстно подчинен только справочнику, который является справочником внешнего контекста (СВК) данной программы **). Ясно, что при таком подходе к определению программы не надо вводить в язык специальную конструкцию для вызова программы, также не надо вводить ограничения на типы параметров (кроме ограничений, связанных с временем жизни типов). Для вызова про-

---

*) Внутри одной задачи можно организовать новую задачу.

**) Кроме того, каждая программа контекстно подчинена постоянно существующей структуре, играющей роль стандартного (предопределенного в языке) контекста.

граммы и передачи параметров можно пользоваться обычной конструкцией вызова процедуры.

Таким образом, мы ввели внешний контекст не только для процедуры активности пользователя, но и для обычной процедуры. Если СВК программы является СВК пользователя, то программе доступен весь архив этого пользователя. Программу можно связать со справочником, через посредство которого доступна только часть архива, т. е. аналогично тому, как в случае обычных процедур существует средство управления доступной частью контекста, так и в случае программ можно управлять размером доступной части архива.

В целом рассмотренный подход дает такие практические следствия:

1. Средствами ЯВУ можно полностью описывать управление архивом и межпрограммное взаимодействие.

2. Совокупность из изображения программы и внешнего контекста повторяет уже известную в ЯВУ совокупность из изображения процедуры и ее глобального контекста.

В обоих случаях используется обычный языковый механизм распределенного контекста, лежащий в основе защиты. В результате полезные свойства этого механизма автоматически переносятся и на процессы обработки архивных объектов, т. е. защита между программами, правильное опознание имен в архиве при дублировании имен, защита между пользователями, возможность структурирования больших программных проектов с защитой от несанкционированного доступа и т. д.

Можно сказать, что расширение сферы применения фундаментальных языковых понятий контекста, контекстного именования и управления временем жизни дает возможность описать все этапы обработки информации пользователем на едином ЯВУ. Одно из практических следствий этого подхода состоит в том, что такой язык может не только использоваться для программирования БПК, взаимодействующих с архивом, но также выполнять в системе роль процедурного языка управления заданиями.

8.5. Пример. Синтаксические и семантические вопросы, связанные с управлением внешними объектами, более подробно рассмотрены в гл. 3. Здесь введем только нотацию для символьного обозначения внешнего объекта и дадим один пример работы с архивом программ. В программе элемент СВК обозначается с помощью конструкции ⟨внешнее имя⟩, имеющей вид *//идент*, где *идент* — идентификатор. Если элемент справочника именует другой справочник, то по аналогии со структурами элемент последнего справочника обозначается двухслоговым именем: *//идент*1 *//идент*2, где *идент*1 — идентификатор элемента первого справочника, а *идент*2 — идентификатор элемента второго справочника. В общем случае имя

может быть многослоговым, что соответствует иерархической структуре внешнего контекста.

Поскольку в языке ДАТД введены средства обработки внешних объектов, в том числе именование таких объектов, то изображение типа и реализации может быть помещено в архив, а в контекстной приставке можно использовать имена типов, хранящихся в архиве. Тем самым в языке даны средства раздельной компиляции и есть возможность хранить в архиве несколько вариантов реализации одного и того же модуля. Динамическая генерация экземпляра модуля (т. е. процедуры, структуры, а в общем случае — модульной конфигурации) производится с помощью генератора, в котором можно указывать список экземпляров структур, включаемых в глобальную контекстную цепочку вновь создаваемого экземпляра раздельно компилированного модуля.

В общем случае модель раздельно компилируемой процедуры имеет вид:

проц (*x1*, *x2*, . . .)
(**контекст огр** *x1*#//*m1*, *x2*#//*m2*, . . . **для**
проц (. . .) (. . .))

Модель раздельно компилируемого составного объекта или модульной конфигурации можно представить так:

проц (*x1*, *x2*, . . .)
(**контекст огр** *x1*#//*m1*, *x2*#//*m2*, . . . **для ген** . . .)

Вызов первой процедуры с параметрами, являющимися экземплярами секций контекста нужного типа, выдает процедурный объект, а вызов второй процедуры — внешнюю часть модульной конфигурации. Поскольку идентификаторы *x1*, *x2*, . . . используются тривиальным образом и только один раз, их можно опускать и изображать текст программы-процедуры в виде:

контекст огр//*m1*, //*m2*, . . . **для проц** (. . .) (. . .),

а текст программы-генератора составного объекта в виде:

контекст огр//*m1*, //*m2*, . . . **для ген** . . . .

В традиционной терминологии это случаи процедуры и пакета, раздельно компилированных в контекстах //*m1*, //*m2*, ...

Теперь рассмотрим пример на языке ДАТД, когда тип пакета, аналогичного описанному в § 6, и процедура генерации пакета являются внешними объектами.

Пусть //*пак* — внешнее имя типа пакета, а изображение типа имеет такой же вид, как указано в § 6. Далее, пусть в контексте пользователя есть две процедуры генерации пакета данного типа с именами //*ген1* и //*ген2*, задающие разные реализации пакета. Эти

процедуры программируются по аналогии с процедурой *генпак* из § 6 с тем отличием, что тип пакета теперь упоминается с помощью внешнего имени *//пак*. Теперь запрограммируем процедуру, которую в традиционной терминологии можно рассматривать как компилируемую в контексте пакета. Причем устроим программу так, чтобы при вызове этой процедуры можно было динамически выбирать либо реализацию *//гп*1, либо *//гп*2 (что уже является нетрадиционным для ЯВУ). Пусть программа генерации этой процедуры имеет имя *//р* и вид *)

проц (*конкр*)
(**контекст огр** *конкр*##*//пак* **для**
проц (. . .) (. . .*созд*. . .*оп*. . .))

Здесь процедура, компилируемая в контексте *//пак*, вложена в основную программу. Вызов этой процедуры с условным выбором одной из реализаций можно изобразить так:

//р (**если** условие **то** *//гп*1 **иначе** *//гп*2 **все**) (параметры)

В общем случае можно описать программу, которая сначала динамически формирует конфигурацию взаимодействующих модулей (процедур и пакетов) с динамическим выбором нужных версий (вариантов) их реализации, а затем активирует выполнение логически головного модуля. Подобные возможности языка можно назвать средствами динамического управления модульной конфигурацией. Отметим, что распространенные ЯВУ, как правило, не имеют таких возможностей работы с архивом модулей. Приходится использовать другой язык — язык редактора связей или язык управления ОС или СП. В свою очередь в языках управления, как правило, нет средств динамического выбора модулей, что делает данную задачу трудно разрешимой.

Еще одна отличительная особенность рассмотренного подхода состоит в следующем. Контроль типа секции глобального контекста (с помощью квалификации) гарантирует, что, если идентификатор, локально описанный в программе А, случайно совпадает с глобальным идентификатором, используемым в программе В, то при динамическом вызове В из А не произойдет совмещение этих разных по смыслу имен. Данное свойство является важным элементом поддержки модульного программирования.

8.6. Справочник внешних связей файла. По аналогии со справочником внешнего контекста программы можно ввести справочник внешних связей файла (СВС). Для этого вводится средство (рассмотренное в гл. 3), позволяющее связать простой файл с некоторым

*) Здесь предполагается, что внешним контекстом этой программы является контекст пользователя.

справочником. При этом предполагается следующая техника использования совокупности из файла и справочника. Элементы СВС именуют внешние объекты, с которыми логически связан данный файл, т. е. логические связи реализуются с помощью файловых ссылок. В информационном содержимом файла все ссылки из данного файла на другие объекты кодируются относительно (или в контексте) СВС. Конкретно, ссылку из файла А на объект В можно кодировать в виде обозначения того элемента СВС файла А, который именует объект В.

9. Свойства модулей

Модулем здесь назовем процедурный или составной объект *) (ниже это определение уточнено). Как видно из вышеизложенного, модуль может иметь достаточно сложную внутреннюю организацию: процедура может быть связана контекстными связями со структурами, структура может быть связана с другими составными и процедурными объектами, образуя в сочетании с ними то, что принято называть пакетом процедур обработки АТД. Когда надо подчеркнуть это обстоятельство, будем применять термин модульная конфигурация. Пользователю модуля, как правило, известна только внешняя спецификация модуля, т. е. тип процедуры или составного объекта, но неизвестна, а главное, недоступна для обработки внутренняя структура модуля или реализация внешней спецификации.

Использованное выше понятие динамического управления модульной конфигурацией (см. последний пример § 8) введено, чтобы отметить тот факт, что язык позволяет программировать динамический выбор (вычисление) необходимой реализации модуля заданного типа, или выбор реализации отдельных элементов конфигурации. Эти реализации могут храниться в архиве, что роднит формирование модульной конфигурации с традиционным понятием сборки программ. Но, как отмечалось, в отличие от традиционных систем, где на специальном языке, например, на языке редактирования связей, описывается статическая конфигурация, здесь на обычном процедурном ЯВУ можно программировать формирование конфигурации как процесс создания объекта сложной структуры. При этом можно использовать все динамические возможности, присущие универсальному языку. В частности, эти средства могут оказаться полезными при программировании систем построения пакетов прикладных программ, одной из задач которых является

*) В Эль-76 модулем называются структуры, используемые для организации пакетов. Учитывая особенности реализации, они вынесены в особый класс объектов.

динамическое формирование (синтезирование) программы из архивных модулей.

Введенные выше определения модуля, разумеется, слишком схематичны. В практике программирования под модульностью обычно понимается набор характеристик:

— возможность обработки модуля регламентируется его внешней спецификацией;

— модуль может иметь скрытые объекты, полностью недоступные пользователю;

— модуль может выдавать в контекст пользователя объекты абстрактных типов, внутренняя структура которых известна и доступна для обработки только внутри модуля;

— взаимодействующие модули могут иметь частично или полностью непересекающиеся глобальные контексты (исключая, разумеется, стандартный контекст);

— модули независимы один от другого в отношении распределения и использования локальных ресурсов;

— модули могут объединяться, образуя новый защищенный модуль;

— модули могут быть полностью независимы один от другого в отношении обработки ошибок;

— внешняя спецификация модуля отделена от его реализации; может быть описано несколько вариантов реализации; спецификация и описания реализаций могут раздельно компилироваться и храниться в архиве.

Сделаем замечание относительно обработки ошибок (более подробно см. [1]). Для обеспечения модульности в части определения составных или сложных действий недостаточно только традиционных структур управления последовательностью действий. В процессе выполнения действий могут возникнуть те или иные ошибки. В этом отношении некоторые распространенные языки неполны. Семантика языка обычно определяет, что такое правильная программа, и тем самым косвенно определяет, что иные действия приводят к возникновению ошибки. Дальнейшее же остается неопределенным, и, соответственно, в языке нет средств анализа ошибки и средств ее обработки.

С другой стороны, для программирования систем необходимы средства, позволяющие программе не терять управление вычислительным процессом даже в том случае, если в ходе счета обнаружились ошибки, а также средства, позволяющие выявить модуль, где возникла ошибка и, если возможно, нейтрализовать ее. Кроме того, в процессе сопровождения и развития систем часто возникает ситуация, когда в работающий комплекс вводится новый или модернизированный модуль, содержащий новые ошибки. В этом случае нужно иметь возможность защитить работающую часть комплекса

от последствий этих ошибок, локализовать неисправный модуль и выдать сообщение или нейтрализовать ошибку. Иными словами, важным элементом модульной организации программы является независимость модулей в части обработки ошибок. Эти соображения говорят за то, что в языке необходим механизм модульной диагностики и обработки ошибок. Такой механизм рассмотрен в [1].

10. Функциональные возможности языка

С точки зрения применения языка свойство полноты и однородности важно потому, что оно определяет, насколько широк диапазон конкретных приложений, где можно использовать преимущества, которые дают базовые механизмы языков. Иными словами, это свойство определяет функциональные возможности языка программирования с позиций его практического применения. Подчеркнем, что при этом обязательно подразумевается, что язык должен быть устроен так, чтобы в конкретных приложениях не надо было жертвовать основными особенностями ЯВУ, т. е. контролем вычислений и защитой объектов.

Данная работа основывается на том, что базовые механизмы ЯВУ, будучи, во-первых, достаточно полно и в достаточно общем виде отражены в конструкциях конкретного языка программирования и, во-вторых, эффективно реализованы, позволяют охватить спектр приложений, который в работе условно назван программированием систем. Можно считать, что в этом спектре приложений важно, обеспечивает ли язык средства поддержки разработки модульных программ и средства управления динамической обстановкой или коротко — средства поддержки *модульности* и *динамического управления*.

В схеме 1 в правой колонке перечислены конкретные возможности ЯВУ, понимаемые как средство поддержки модульности и динамического управления. В средней колонке перечислены базовые механизмы ЯВУ, а в левой — механизмы, порожденные путем сочетания базовых элементов. К схеме надо сделать ряд пояснений:

1) Наряду с базовыми механизмами упомянут принцип универсальной применимости базовых средств (п. 2.4), который в данной работе используется, во-первых, при введении новых (порожденных из базовых) механизмов ЯВУ, например, средств управления внешними объектами. В этом случае расширение возможностей языка достигается за счет расширения сферы применения базовых механизмов. Во-вторых, этот принцип требует, чтобы не ограничивались возможности обработки таких объектов, как процедуры и пр. Это даёт вклад в средства поддержки динамического управления.

2) В схеме не проведены конкретные связи от базовых механизмов к возможностям языка, обусловленным этими механизмами.

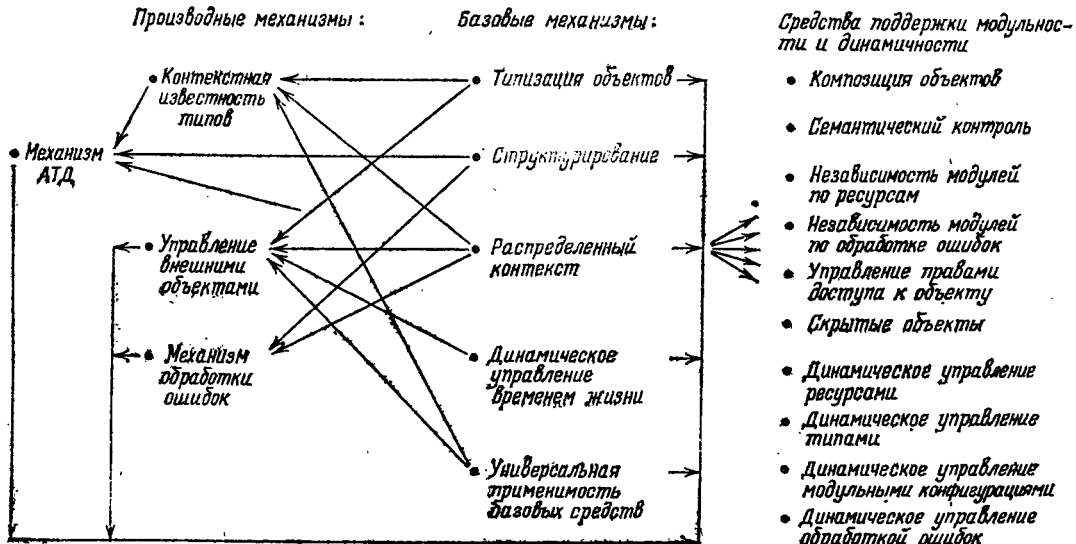


Схема 1. Связь базовых и производных механизмов ЯВУ и функциональных возможностей языка

Дело в том, что практически каждый базовый механизм вносит тот или иной вклад в возможности языка, поэтому отдельные связи заменены одной общей связью. Что касается связей между базовыми и порожденными механизмами, то по аналогичной причине на схеме отмечены только первостепенные связи.

3) Алгоритмы программируются в терминах объектов и, следовательно, не описывают физическое распределение ресурсов. Акты создания и уничтожения объектов могут быть не связаны жестко со скобочной (фразовой) структурой текста программы. Объект можно создавать динамически, и его время жизни может быть больше или меньше времени выполнения фразы, внутри которой он создан. В частности, таким образом были получены внешние объекты, время жизни которых может быть дольше времени выполнения программ, создающих и использующих эти объекты. Как видим, этот метод управления временем жизни дает определенный вклад в расширение возможностей языка в части динамики управления объектами, а также непосредственно приводит к средствам управления архивом. В схеме 1 внешние объекты особо не выделяются, но подразумевается, что к этому классу объектов применимы все перечисленные возможности.

4) Если принять во внимание реализационные аспекты, то оказывается, что понятие ресурса все-таки приходится учитывать при разработке больших программ, причем динамическая системная стратегия распределения ресурсов оперативной и внешней памяти лучше согласуется с требованиями модульности, чем статическая. Тот факт, что локальные объекты модуля автоматически уничтожаются при завершении его работы, ориентирует пользователя на естественный стиль программирования, когда внутренние объекты модулей не помещаются в глобальное окружение, а локализуются в соответствии с логикой их использования и модульной структурой программы. Это дает определенный вклад в модуляризацию программы и одновременно обеспечивает экономное использование ресурсов памяти.

В отличие от этого системная статическая стратегия распределения ресурсов плохо согласуется с модульностью, так как вынуждает для больших программ заводить глобальные массивы, общие для нескольких модулей, но используемые ими для своих внутренних нужд. В этом случае явным образом программируется динамическое перераспределение общего ресурса. Это является источником ошибок и по существу означает, что вместо программирования в терминах объектов приходится описывать алгоритмы в терминах ресурсов.

5) Средства параллельной обработки относятся одновременно и к средствам динамического управления вычислительными ресурсами, так как позволяют полнее использовать многопроцессорную

систему, и к средствам поддержки модульности, так как обеспечивают независимость процессов в отношении порядка их выполнения.

6) Динамическое управление модульными конфигурациями было рассмотрено выше в § 9. Добавим, что сюда же надо отнести средства динамического вызова программ-процедур из архива без предварительной статической редакции связей.

7) Механизм динамического управления типами удобен тем, что сохраняя одно из важнейших свойств ЯВУ — семантический контроль правильности обработки информации, он одновременно дает гибкость программирования. Кроме того, как отмечалось в § 3, с точки зрения концептуальной однородности языка принцип динамического управления типами лучше сочетается с принципом контекстной известности типа, чем принцип статического контроля. В случае языка класса Д не требуется указывать тип значения переменной. В частности, для обработки объектов абстрактных типов нужно знать только имена процедур обработки (см. пример описания пакета в § 5). Для создания объекта и проверки типа в пакете должны быть предусмотрены соответствующие примитивы генерации и предикат проверки. Можно сказать, что требованиям модульности лучше всего удовлетворяет динамическая концепция, тогда как статическая приводит к определенным языковым аномалиям.

Наконец, взяв принцип динамического контроля за исходную концепцию, можно далее ввести в язык элементы статики, которые будут использоваться при программировании только в тех случаях, когда статические соотношения между объектами естественно вытекают из природы описываемого алгоритма. Примеры такого подкода рассмотрены в § 7. В отличие от этого, если за основу берется статическая концепция, то при программировании алгоритмов, динамичных по своей природе, приходится специально приспосабливать программу к этому языковому ограничению, либо просто отключать контроль типов. Это можно рассматривать как усложнение программирования и источник ошибок, именно в случае систем повышенной сложности.

Резюмируя, отметим, что ЯВУ с названными средствами поддержки модульности и динамического управления при условии эффективности рабочих программ, написанных на этом языке, можно рассматривать как ЯВУ, универсально применимый для программирования систем в том смысле, что по объему функциональных возможностей он обеспечивает нужды профессионального программирования, и на практике не требуется применять отключение семантического контроля и прочие методы ассемблерного программирования. Одна из основных целей работ по Эль-76 состояла в построении конкретного языка, обладающего свойством универсальности в указанном смысле.

11. Функциональные возможности языка и эффективность программ

Разнообразие методов обработки информации, используемых в рамках БПК, может быть большим: расчеты методами численного анализа, вызов программ из архива, обработка текстов, компиляция, управление архивом или базой данных, динамический заказ и освобождение памяти, параллельная обработка, обнаружение и нейтрализация ошибок, обработка в реальном масштабе времени. Чтобы обеспечить эти нужды, язык программирования систем должен наряду с эффективностью обладать достаточной функциональной полнотой. Отсюда не следует, что язык должен механически объединять разнородные методы (так как объем перебора слишком велик). Сочетание названных в § 10 средств поддержки модульности и управления динамической обстановкой может дать язык, обладающий необходимой степенью универсальности, сохраняя при этом основные черты ЯВУ и семантический контроль вычислений *).

Но для этого каждый отдельный механизм должен быть представлен в языке в достаточно общем виде. Практическому воплощению такого подхода препятствует архитектура ЭВМ. Если она недостаточно хорошо согласована с основными языковыми механизмами, то значительная часть номинальной производительности машины может пойти на компенсацию рассогласования и рабочие программы будут иметь низкую эффективность. Поэтому, как уже отмечалось, в случае традиционных архитектур ограничивают функциональные возможности языка, чтобы достичь удовлетворительной эффективности рабочих программ по крайней мере для класса приложений.

Ограничение возможностей языка приводит к тому, что он не обеспечивает всего спектра нужд БПК и приходится идти на снижение уровня и, соответственно, надежности программирования и использовать ассемблер как универсальный инструмент программирования систем. В п. 2.4 отмечалось, что в последнее время нашли применение (особенно в системном программировании) такие языки, которые можно было бы поместить между ассемблерными языками и ЯВУ. Их изобразительные средства похожи на принятые в ЯВУ, но есть и различного рода ассемблерные характеристики: контроль типов либо не производится, либо есть возможность его отключать, разрешен выход в ассемблерный язык и пр. (подробно см. § 13). Поскольку эти характеристики, в частности, отсутствие контроля, снижают надежность программы, мы будем условно причислять

*) В контексте данной работы сохранение основных черт ЯВУ является важным условием решения задачи разработки языка программирования систем.

такие языки (точнее — их машинно-ориентированные средства) к ассемблерным средствам программирования (АСП).

Несмотря на введение ограничений, не всегда удается удовлетворить требованию эффективности рабочих программ. Данные, показывающие снижение эффективности при переходе от Ассемблера к ЯВУ, уже были указаны в § 1.

12. Ограничения и ненадежные конструкции языков

Различия между механизмами ЯВУ и возможностями среды реализации сводятся в основном к следующему:

— структуризация и типизация объектов, а также контроль их обработки в ЯВУ в противовес хранению неструктурированной битовой информации и неконтролируемым вычислениям в ЭВМ*);

— механизм распределенного контекста и контекстная защита объектов в ЯВУ в противовес механизму адресации линейной (или в лучшем случае разбитой на листы) памяти в ЭВМ и методам именования архивных объектов, применяемым в ОС;

— недостаточная поддержка со стороны ОС динамического управления временем жизни объектов.

Для достижения удовлетворительной эффективности (например, при научно-технических расчетах) в язык вносится ряд ограничений:

— статика типов,

— ограничения на обработку некоторых классов объектов (процедур, пакетов, ситуаций),

— ограничения динамики создания/уничтожения объектов.

Кроме того, распространенные ЯВУ неполны в части средств обработки динамических ошибок и средств управления объектами архива.

Приведем пример, иллюстрирующий природу ограничений. Чтобы реализовать универсальные средства обработки процедурного объекта, в СП надо ввести объект, содержащий адресную информацию о коде процедуры и ее статическом окружении (представитель процедуры), и реализовать пересылку этого объекта в памяти. Отсюда вытекают следующие трудности: (а) возникает известная проблема контроля «зависших указателей», (б) требуется реализовать общий случай сохранения и коррекции контекста при входе в процедуру и выходе из нее.

Если же представитель процедуры не пересылается, то первая проблема ликвидируется (в части процедур), а процедурный механизм можно реализовать в ограниченном виде, чтобы упростить

*) Традиционные виды аппаратного контроля четности и пр., разумеется, важны, но их семантический уровень ниже уровня языкового контроля.

реализацию и повысить эффективность на машине с традиционной архитектурой, не имеющей соответствующей аппаратной поддержки. Например, если исключить механизм формальных процедур, то коррекция окружения при вызове процедуры в худшем случае сводится к упрятыванию и установке нового содержимого одного регистра, тогда как общий механизм распределенного контекста, описанный в § 3, требует полной коррекции окружения при входе в процедуру, причем области, относительно которых производится базирование, могут находиться вне стека активаций процедур.

Что же касается возможности вызова программ из архива, а тем более передачи их в качестве параметров, то такие средства в определении распространенных ЯВУ вообще отсутствуют, так как здесь требуется соответствующая поддержка не только со стороны аппаратуры, но и со стороны ОС.

Ограничения динамики сужают область применения ЯВУ, особенно в части программирования систем, взаимодействующих с человеком. В связи с этим можно привести такие неформальные соображения.

Будем рассматривать систему, состоящую из человека (или другого объекта с недетерминированным поведением) и программы, взаимодействующей с человеком. Особенность этой системы состоит в том, что присутствие в ней человека вносит в поведение всей системы элементы недетерминированности, поскольку действия человека статически непредсказуемы *). Можно считать, что в этом заключается причина динамичности систем, взаимодействующих с человеком, или по крайней мере их компонентов, непосредственно ответственных за взаимодействие, а также причина того, что для программирования подобных систем требуются средства управления динамической обстановкой. Один из примеров динамики такого происхождения уже разбирался в § 8, когда рассматривалось взаимодействие с внешними объектами. То же самое можно сказать относительно динамики управления ресурсами и типами объектов.

Для ряда БПК в той или иной части требуется динамический контроль типов по той причине, что тип обрабатываемых данных может быть неизвестен при компиляции программы. Наиболее выразительным примером является операционная система. Для этой программы характерно то, что она осуществляет обработку программ пользователя, обработку областей памяти, занятых объектами пользователя, распределением памяти для объектов пользователя и т. д. Типы объектов пользователя, спецификации параметров программ пользователя могут быть по существу неизвестны при компиляции ОС. Эта особенность имеет место и для других систем, которые, вы-

*) В противном случае можно было бы статически предсказать будущие события и, если так можно выразиться «откомпилировать» историю мира как единую программу.

полняя приказ пользователя, должны манипулировать с его данными и программами. С другой стороны, как уже говорилось, языки класса С постулируют ряд статических свойств объектов. Эти свойства должны иметь место вне зависимости от природы алгоритма. Такое требование ограничивает применимость языка.

Чтобы преодолеть перечисленные сложности, при программировании систем частично или полностью применяются АСП. Соответственно системные языки являются многоуровневыми и наряду с традиционными конструкциями ЯВУ содержат арсенал средств неконтролируемого ассемблерного программирования. Иерархическая структура языка является, разумеется, важным инструментом построения сложного из простого. Однако можно отличать естественную иерархию, когда понятия промежуточных уровней, образно говоря, лежат «на прямом пути» между нижним уровнем и верхним *), от вынужденной иерархии, возникающей из-за того, что разработчик языка должен приспосабливать язык к наперед заданной аппаратуре.

Здесь надо учесть, что особенности архитектуры традиционных ЭВМ во многом объясняются экономическими мотивами, объективными для состояния технологии в 40-х и 50-х годах, а не ориентацией на описание алгоритмов (это обстоятельство обсуждалось во введении). Поэтому структуру современных языков системного программирования можно отнести к разряду «вынужденной» иерархии.

Формы, с помощью которых АСП представлены в языках, существенно различаются:

— явное использование регистров (GSL, GPL, PL/11, Chill) [95];

— неконтролируемое преобразование типов (Мэри [98], Эвклид, Ада, Си [100]);

— бестиповые указатели (LIS [96], ПЛ/1, GSL, PS440 [97]);

— арифметика над указателями (Си, Chill, Блисс [92]);

— неконтролируемые вариантные записи (Си, Паскаль **), Sydec [95]);

— машинно-ориентированные модули (Эвклид);

— спецификации реализации (Ада, Chill).

В целом эти средства можно охарактеризовать как ассемблерные в том смысле, что они допускают бесконтекстную трактовку информации, т. е. обработку, не связанную с типом данных и дос-

*) Здесь можно провести аналогию с методикой иерархической разработки программ в структурном программировании.

**) В описании языка Паскаль [99] не определено, осуществляется ли динамический контроль типа вариантной записи. Что же касается реализаций, то в большинстве из них такой контроль не проводится.

тупным пространством имен. Как видим, в каждом отдельном случае расширение возможностей достигается за счет того, что, жертвуя концептуальным единством языка, в него вносятся ненадежные конструкции. Вытекающее отсюда уменьшение надежности программ следует признать основным недостатком такого подхода.

Приведем взятый из работы [54] список ошибок, характерных для программ на языке ПЛ/1 и Фортран в системе 370, а затем проведем сравнение с МВК Эльбрус. Эти примеры иллюстрируют влияние архитектуры на надежность языка и программ. Основная неприятность состоит в том, что семантически неправильная конструкция выполняется без какой-либо диагностики, а последующее внешнее проявление ошибки может не иметь ничего общего с ее изначальной причиной.

Приводимый список представителен не только с точки зрения анализа ПЛ/1, но и других языков, так как реализация на традиционной архитектуре формально ставит их в аналогичные условия *):

1. Использование значения неинициализированной переменной. Эта распространенная ошибка имеет отношение ко всем ЯВУ независимо от того, есть ли в них АСП. Для ее обнаружения целесообразно иметь соответствующую аппаратную поддержку.

2. Выход индекса за границу массива. В реализациях ЯВУ контроль часто отключается из-за связанных с ним накладных расходов.

3. Программа (модуль) обращается через посредство указателя к недоступным ей данным другой программы (другого модуля) или обрабатывает код как данные.

4. Программа обращается через посредство указателя к области памяти, от которой она ранее отказалась (проблема «зависших указателей»). К пп. 3 и 4 можно добавить случай, когда при статическом распределении памяти два модуля одновременно используют один и тот же участок общей области в разных целях, причем каждый из них предполагает, что он единолично владеет данным участком.

5. Программа считывает в переменную запись файла, имеющую иную структуру, нежели предписано описанием переменной.

6. Ошибка в вычислении адреса может возникнуть, если элемент памяти, распределяемой компилятором, меньше минимального адресуемого элемента памяти машины. Например, в ПЛ/1 битовые строки не обязательно начинаются с начала байтового элемента памяти, но память в системе адресуется с точностью до байта. Если процедуре в качестве параметра передается адрес начала строки,

*) Есть неформальное отличие некоторых ЯВУ от ПЛ/1. Эти языки устроены так, что хотя АСП в них присутствуют, но их использование «затруднено».

то последующее обращение из процедуры к этой строке может произойти неверно.

7. Число аргументов процедуры, их типы и размеры не соответствуют формальным спецификациям. Здесь важно то, что поскольку в памяти не хранится информация о типе и размере, последующая обработка параметров в процедуре может не сразу привести к возникновению ошибки.

8. Два модуля содержат несогласованные описания одной и той же глобальной переменной. Эта ошибка, как и предыдущая, связана также с тем, что редактор связей не контролирует соответствие типов при сборке модулей. Аналогичная ошибка может возникнуть и в языках класса Ады, так как в них есть явное назначение абсолютных адресов и двум переменным разных типов или двум подпрограммам с разной спецификацией параметров может быть назначен один и тот же адрес.

Дополним этот список известными примерами из Фортрана.

9. Контроль типов элементов общих областей практически отсутствует. В сущности, в языке просто отображена восходящая к сборке ассемблерных программ техника наложения общих областей путем совмещения адреса их начала без контроля соответствия структур этих областей. Отсюда возможны разнообразные ошибки обусловленные тем, что статический контроль типов отсутствует. Поскольку аппаратура не распознает тип и размер данных, динамический контроль также не осуществляется.

10. Вещественные числа могут восприниматься как целые и наоборот. Например, программа:

```
EQUIVALENCE (x, i)
x = 3.14
i = i — 1
WRITE (. . .) i
```

вообще говоря, является ошибочной, причем обнаружить ошибку при компиляции сложно. Если в машине различаются представления целых и вещественных чисел и аппаратура не распознает тип данных, то при выполнении программы произойдут две ошибки: целочисленное вычитание целого из вещественного числа и печать вещественного числа как целого.

11. Изменение значения константы. Поскольку в традиционных архитектурах нет соответствующей поддержки языкового механизма передачи параметров, в реализациях часто применяется способ, при котором параметры всегда передаются по адресу, в частности, и тогда, когда параметр является константой, например, числом. Для этого отводится ячейка, куда помещается число, а в процедуру передается адрес ячейки. Кроме того, что это замедляет программу (вместо одного — два считывания из памяти при обращении к па-

раметру), оказывается, что можно изменить значение такой константы. А. Н. Терехов приводит следующий пример:

```
CALL F (I)
I = 1
WRITE (. . .) I
STOP
END
SUBROUTINE F (J)
J = 0
RETURNE
END
```

Эта программа напечатает не 1, а 0, так как процедура F записывает в стандартную ячейку, отводимую транслятором для константы 1, число 0.

В МВК Эльбрус контроль этих ошибок осуществляется аппаратно. Рассмотрим по отдельности каждую ошибку в том же порядке, как они перечислены для традиционных архитектур.

1. В качестве начального значения переменных без инициализации используется значение типа «пусто». Попытка обработки этого значения в операции приводит к аппаратно диагностируемой ошибке.

2. Производится аппаратный контроль выхода индекса за границы размера массива.

3. Производится общий контроль «границ» доступной области данных (контекстная защита). Это свойство вытекает из аппаратной реализации процедурного механизма и контроля границ составных объектов. В свою очередь реализация этих механизмов формально следует из тегированной памяти (есть теги процедур и составных, поэтому надо реализовать соответствующие операции и контроль).

4. Механизм математической памяти устроен так, что при обращении к ранее освобожденной области выдается диагностика об ошибке. Далее, вместо «ручного» распределения общей памяти можно использовать механизм динамических генераторов, являющийся основным в системе Эльбрус.

5. В операциях производится аппаратный контроль стандартных типов и размеров данных.

6. В МВК Эльбрус возможна адресация и соответственно формирование указателя с точностью до бита.

7. Для контроля этой ошибки используется тегированная память, как и в случае 5. Кроме того, используя аппаратные средства, можно отделить область параметров от области локальных. Тогда попытка обращения к формальному параметру, для которого нет соответствующего фактического, приведет к аппаратно диагностируемой ошибке.

8, 9, 10. Для контроля этих ошибок используется тегированная память, как и в п. 5.

Проблему, проиллюстрированную в п. 10 и связанную с некорректным использованием эквивалентности переменных, можно решать в реализации Фортрана двумя способами.

Во-первых, в системе команд есть набор целочисленных операций, включая операцию записи, динамически контролирующих принадлежность операндов и результата к множеству целых. Если операцию вычитания в третьей строчке компилировать в такую команду, то выполнение оператора приведет к аппаратно-диагностируемой ошибке.

Во-вторых, можно расширить семантику Фортрана и считать, что, если задана эквивалентность переменных, то тем самым неявно определена одна переменная, тип которой может быть любым числовым типом. Тогда операцию в третьей строчке надо компилировать в полиморфную операцию вычитания, и переменной будет присваиваться вещественное число. Процедура вывода может по тегу анализировать тип числа и выводить его на печать в соответствующем виде. Аналогично решается проблема, возникающая в традиционных системах из-за эквивалентности переменных одинарной и двойной точности.

11. В Фортране-Эльбрус используются специальные команды обработки параметров и тегированная память. В результате в процедуру можно передать число, а не его адрес. Это, во-первых, приведет к одному обращению в память вместо двух и, во-вторых, программа будет работать верно и отпечатает число 1.

13. Влияние архитектуры на разработку систем программирования

При разработке систем программирования (СП) для ЯВУ все большее внимание уделяется вопросам построения СП, поддерживающих разработку БПК на всех стадиях их жизненного цикла: написание — автономная отладка — комплексная отладка — документирование — сопровождение — модернизация. Такие СП с расширенными возможностями иногда называют системами поддержки программирования (СПП). Особо выделяются такие аспекты СПП: поддержка разработки отдельных модулей и сочленения их в БПК, обеспечение средств анализа текущего состояния программного проекта и истории его создания (наличие версий программного комплекса, документация, результаты тестирования), наличие достаточно полного набора инструментальных средств для различных способов обработки программ, таких как отладчики, редакторы и пр.

Еще одно важное требование состоит в том, чтобы все общение СПП с пользователем велось в символьных терминах обрабатываемой программы. Эти требования основаны на том соображении, что

наличие интегральной и полной среды поддержки разработки БПК является необходимым условием получения экономического эффекта, заложенного в применении ЯВУ для программирования этих комплексов. В материале «Требования к среде поддержки программирования на языке Ада» [101] отмечается: «В прошлом и во многих случаях текущей практики концепция систем поддержки недостаточно ясно осознана. Существующие инструментальные средства не представляют целостной системы, а их совокупность неполна».

13.1. Традиционный подход. Кратко проанализируем причину сложности разработки СПП на базе традиционных архитектур. Объектами обработки СПП являются программа, точнее, совокупность модулей, хранящихся в архиве, и ряд других внешних объектов, имеющих отношение к программе (исходные тексты модулей, документация и пр.). С точки зрения СПП объекты этой совокупности, которую будем называть *программной конфигурацией*, находятся в определенной логической взаимосвязи, т. е. программная конфигурация имеет некоторую логическую структуру.

Далее, если проанализировать обработку программы, например, на этапах отладки, раздельной компиляции, то видно, что собственно программа или ее отдельный модуль тоже обрабатываются как структурированные объекты, структура которых определяется в терминах ЯВУ. Можно различать текстовое представление программы и кодовое (внутреннее) представление программы. В ходе выполнения программы возникает еще один объект — память выполняемой программы. Этот объект также обрабатывается СПП как структурированный в терминах ЯВУ. В режимах, когда СПП обрабатывает выполняемую программу (например, отладка, аварийная выдача), нужно иметь возможность устанавливать соответствие между элементами структуры памяти и элементами кодового представления, с одной стороны, и элементами текстового представления, с другой стороны, что позволяет вести выдачу в символьном виде, привязанном к тексту исходной программы.

Такая модель работы СПП и структуры обрабатываемых ею объектов является идеализированной. На самом деле в традиционных системах структуры внутреннего представления программной конфигурации (на уровне архива), собственно программы и память выполняемой программы неадекватны их логической структуре. Архив, как правило, имеет плоскую, одномерную структуру и/или нет механизмов, позволяющих непосредственно отобразить логические связи между элементами программной конфигурации. Разумеется, существенную помощь в решении этой проблемы может оказать СУБД. Однако в некоторых проектах СПП [102] высказывается мнение, что использование универсальной СУБД может отрицательно сказаться на эффективности СПП.

Набор требующихся структур и связей здесь ограничен, и, возможно, более целесообразно иметь специализированную СУБД или же строить СПП на основе развитой системы файлов, дающей возможность связывать архивные объекты ссылками, вводить атрибуты для архивных объектов и др. особенности, близкие к тем, которые рассматривались в § 8. С другой стороны, есть примеры СПП, ориентированных на универсальные СУБД [103]. Во всяком случае, отметим, что традиционного плоского архива без файловых ссылок и определяемых пользователем атрибутов недостаточно для организации архива СПП, и в этом случае требуется соответствующая программная надстройка.

Относительно внутреннего представления программы можно сказать следующее. Программа, как известно, является объектом обработки не только для СПП, но для аппаратуры и ОС. Однако в случае традиционных архитектур структура этого объекта на уровне аппаратуры и ОС отличается от той, о которой говорилось выше. Действительно, структура программы и памяти выполняемой программы является стандартом в вычислительной системе, который определяется архитектурой машины и ОС. Поскольку есть семантический разрыв между архитектурой и ЯВУ, то такая же несогласованность будет между стандартными системными структурами, с одной стороны, и структурами исходной программы и объектов выполняемой программы, с другой. Вкратце это выражается в том, что на уровне аппаратуры теряется информация о типах объектов и их структурной и контекстной подчиненности. Чтобы компенсировать рассогласование, нужно создавать дополнительные информационные структуры, определяемые и интерпретируемые конкретной СПП. В целом можно заключить, что проблемы разработки СПП, как и проблемы разработки ЯВУ, вызваны рассогласованием архитектуры и ЯВУ.

13.2. Принципы поддержки СП в системе Эльбрус. Чтобы пояснить возможности системы в части поддержки СП, рассмотрим логические модели объектов, обрабатываемых системой. Эти модели имеют почти непосредственные реализационные аналоги, но в контексте данной работы достаточно проанализировать логическую структуру. Другие концептуальные и реализационные аспекты подробно рассмотрены в работах [2—4, 7—14, 19—21, 28—33].

Среди основных принципов разработки системы Эльбрус выделим два принципа, которые существенно влияют на разработку СП и вместе с тем хорошо согласуются с базовыми механизмами ЯВУ. Первым является принцип типизации и структурирования системных объектов. Его смысл состоит в том, что на уровне системы должна быть сохранена информация о структурных свойствах исходной программы и объектов выполняемой программы (имеется в виду информация о типах и логической структуре). Для этого

структура системных объектов должна быть адекватна структуре объектов языка.

Второй принцип состоит в том, что надо системно реализовать механизм распределенного контекста, так как на нем основана защита, контроль контекстной известности типов и в конечном счете механизм АТД. Поэтому в последующем изложении на первом плане, как и при анализе языка ДАТД, будут стоять вопросы обработки процедурных и составных объектов. Практическое воплощение названных принципов упрощает реализацию языков с механизмом АТД и языков класса Д, упрощает реализацию СП, о которых говорилось выше, так как системное хранение информации о структурных свойствах программы дает готовую информационную базу для работы этих СП.

Аналогия с интерпретирующим автоматом. Существует определенная аналогия между процессом обработки программы в языкоориентированной системе и процессом обработки программы с помощью гипотетических вычислителей, т. е. интерпретирующих автоматов, используемых для описания операционной семантики языков программирования, например, в Венском методе описания (ВМО). Структуры, обрабатываемые системой, можно подразделить на структуру программы (в терминологии ВМО — абстрактную программу), обработка которой ведется с помощью операций (команд) высокого уровня, выбираемых в зависимости от обрабатываемого элемента структуры программы, и структуру памяти выполняемой программы (в терминологии ВМО — состояние управления, внешней среды, хранилищ, счетчика отличительных имен и склада).

Аналогия с гипотетическим вычислителем имеет известные пределы. Хотя модели вычислений, производимых интерпретирующим автоматом и реальной языкоориентированной вычислительной системой, имеют, естественно, много общего, так как в конечном счете отображают семантику языков процедурного класса, при разработке системы Эльбрус потребовалось заново спроектировать структуры объектов. Различия обусловлены разным применением этих вычислителей. Первый ориентирован на формальное описание семантики конкретного языка в терминах символьного выполнения программы. Второй ориентирован на реальное выполнение компилированных программ и на класс языков, в состав которого входят не только традиционные ЯВУ, но также языки с расширенными функциональными возможностями типа ДАТД. Кроме того, он должен обеспечивать поддержку СП. В связи с этим необходимо было учитывать не только традиционные языки, но также обобщения и новые черты процедурных языков в части рассмотренных выше механизмов модульности и динамики. При разработке структур программы и памяти выполняемой программы надо было ориентироваться не на конкретный язык, а выделить и реализовать фундаментальные свойства

программ, написанных на ЯВУ. Кроме того, необходимо было учесть различные режимы обработки программы: (а) режим выполнения, когда процессор, ориентированный на класс процедурных языков, может обрабатывать не специфичное для данного языка символьное, а универсальное кодовое (компилированное) представление программы; (б) режимы обработки программы компонентами СП, когда необходима информация, в том числе — символьная информация, о структурных свойствах программы и памяти выполняемой программы, информация о соответствии кодового и символьного представлений.

13.3. Структурная модель программы — это абстрактная структура данных, конкретизацией которой являются стандартизованные в системе структуры файлов, включая файл объектного кода. Они содержат внутреннее представление программы, обрабатываемой машиной, операционной системой и компонентами СП. В основу построения структурной модели положен принцип статической декомпозиции программы. Этот известный принцип программирования имеет целый ряд полезных следствий с точки зрения удобства обработки программы собственно системой, а также инструментальными компонентами СП.

В соответствии с этим принципом изображение программы можно рассматривать как структуру, состоящую из статических объектов (см. классификацию § 8), создаваемых при компиляции программы и обрабатываемых в таком виде при исполнении программы, а также из п р о о б р а з о в оперативных объектов. Здесь имеется в виду, что в процессе выполнения программы из прообраза объекта создаются (или генерируются) э к з е м п л я р ы объектов. Экземпляры образуют структуру памяти выполняемой программы. В качестве таких прообразов будем рассматривать изображение типа и изображение процедуры, исполнение которых приводит к созданию соответственно типа и процедурного объекта, изображение или текст конструкции генератора составного объекта, исполнение которого приводит к созданию составного объекта *). Прообразы связаны между собой связями, которые также известны статически. Во-первых, это структурные связи, отражающие текстуальную вложенность прообразов, и, во-вторых, прообразы контекстных связей. Последние в результате генерации объектов переходят в соответствующие им контекстные связи между объектами. Чтобы проиллюстрировать этот процесс, рассмотрим прообразы процедуры и составного объекта, изображенные на рис. 4 и 5.

---

*) Уточним, что для составных объектов в стандартных файлах системы Эльбрус хранятся только прообразы генераторов так называемых модульных объектов, которые по реализационным соображениям выделены в отдельный класс составных объектов.

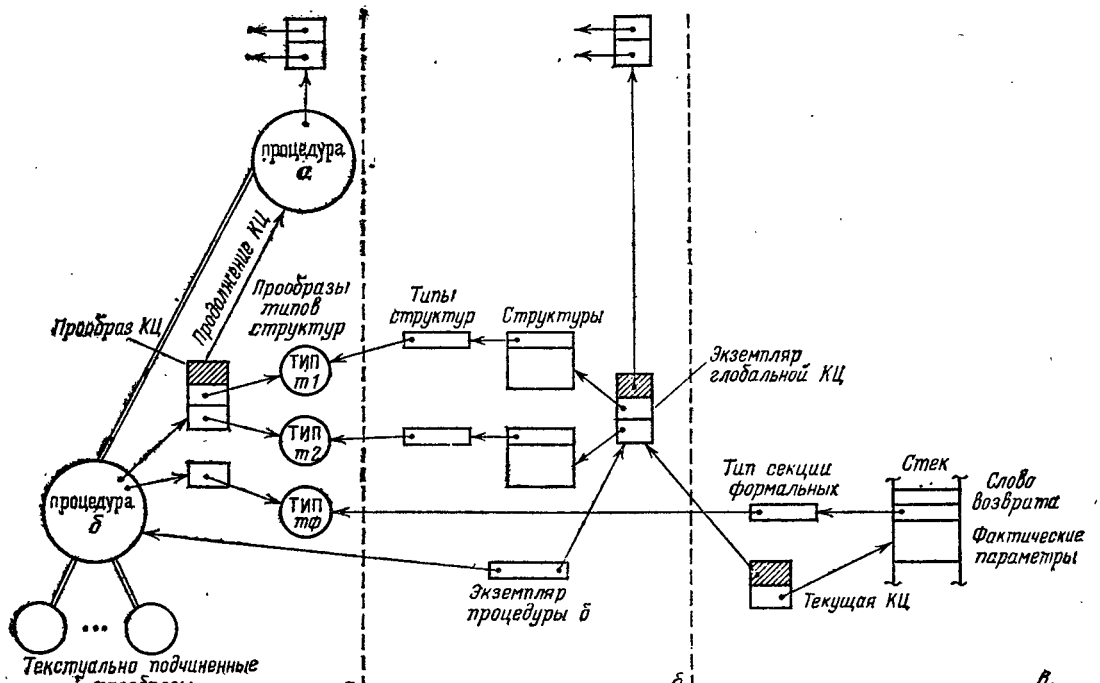


Рис. 4. Структурная модель процедуры, генерация экземпляра и вызов: *а* — прообраз процедуры *б*, вложенной в *а*; *б* — генерация экземпляра *б*; *в* — вызов процедуры *б*. КЦ — контекстная цепочка.

Сделаем предварительные пояснения к рисункам:

1. Структурные модели строятся из узлов, соответствующих прообразам. Узел снабжен набором атрибутов. Атрибутом может быть значение, характеризующее определенное свойство прообраза, или связь (связи) с другим узлом (узлами). Основными атрибутами узла являются: структурные связи с узлами, соответствующими непосредственно текстуально вложенным прообразам (эти связи обозначаются двойной линией); контекстные связи с узлами, соответствующими типам элементов контекстной цепочки данного прообраза; класс генерируемого объекта (сокращенно обозначается «тип»,

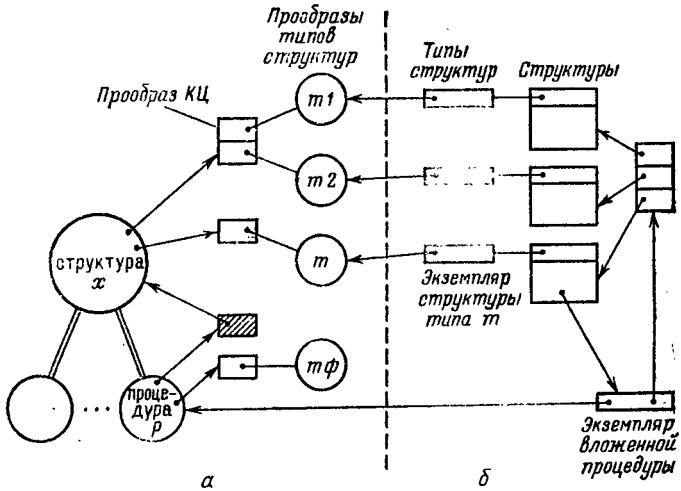


Рис. 5. Прообраз составного объекта и генерация экземпляра: *а* — прообраз структуры с вложенной процедурой; *б* — генерация экземпляра структуры

«процедура», «структура»); символьное имя генерируемого объекта, если оно может быть статически установлено по тексту программы, а в противном случае некоторый способ именования объектов по дереву текстуальной вложенности.

2. Изображению типа, которое встречается в описании типа и из которого создается новый тип, соответствует узел класса «тип» с именем, взятым из левой части описания типа. Для анонимных типов (изображение типа структуры формальных параметров, непосредственное изображение типа в генераторе составного объекта) также заводится узел класса «тип». При исполнении прообраза типа создается новый объект в том смысле, что он отличается от всех других объектов. Способ обеспечения различимости типов относится к реализационной стороне дела и пояснен в [19]. Одним из компонентов типа является прообраз типа (в реализационном отношении эта связь представляется ссылкой из типа). Наличие этой связи объясняется

не языковыми соображениями, а требованиями технологической поддержки СП. Ее смысл будет объяснен ниже.

3. Скрытым элементом объекта является его тип. К этому элементу нет прямого доступа с помощью конструкции выбора, и он неявно используется для контроля типов, а также при выполнении квалификации и операции проверки типа. Этот элемент отмечен только в случае составных объектов выполняемой программы на рис. 1, а на последующих рисунках тип обозначается, только если это нужно по существу изложения.

4. В левой части рисунков изображены структурная модель исходной программы, т. е. прообразы объектов, а в правой — объекты, создаваемые из прообразов. Такое объединение рисунков сделано, чтобы иллюстрировать взаимосвязь между структурной моделью программы и структурной моделью памяти выполняемой программы и показать, что СП может выполнить обратную проекцию объекта памяти на символьное представление прообраза этого объекта *). Это средство необходимо для поддержки символьной выдачи компонентов СП.

5. В отличие от абстрактной модели для интерпретирующего автомата, излагаемая модель, вообще говоря, не предназначена специально для символьного исполнения программы. В первую очередь в модели в непосредственном виде представлено разбиение программы на прообразы типов, процедур и связи между прообразами. Это сделано по той причине, что для компонентов СП и ОС необходима информация именно об этих структурных свойствах программы, т. е. информация о макроструктуре программы.

6. Будем считать, что два объекта А и Б тождественно равны, если А и Б — это один и тот же объект. В моделях это определение используется в основном для того, чтобы определить, равны ли элементы двух объектов. Например, два объекта имеют один и тот же тип, если в графическом изображении этих объектов стрелки, соответствующие скрытым элементам, задающим тип объектов, приводят к одному и тому же объекту.

Прообраз процедуры. На рис. 4, *а* изображена структурная модель такого фрагмента программы

конст *а* = **проц** (. . .) (. . .
конст *б* = **контекст** *e*1#*m*1, *e*2#*m*2
для проц (. . .) (. . .). . .)

Атрибутами прообраза вложенной процедуры *б* являются:

— структурные связи с прообразами, непосредственно вложенными в *б*. Они обозначены двойными линиями, соединяющими *б* с вложенными прообразами;

*) Имеется в виду, что эта проекция — обратная по отношению к генерации объекта из прообраза, которую производит процессор.

— контекстные связи. Они изображаются следующим образом. Столбиком гнезд около узла *б* представлен прообраз глобальной контекстной цепочки, соответствующий контекстной приставке в изображении *б*. Стрелка из каждого гнезда ведет к прообразу типа элемента цепочки. В соответствии с соглашениями § 7 эта связь статически известна. Структурная связь, приводящая к прообразу типа, на рисунке не обозначена. Верхнее заштрихованное гнездо присутствует в том случае, если контекст процедуры явным образом не ограничен. Тогда стрелка ведет к узлу, соответствующему процедуре, контекстная цепочка которой добавляется в начало данной, т. е. эта стрелка указывает на продолжение контекстной цепочки. Вторая контекстная связь ведет из узла к прообразу цепочки контекста, которая строится из секции формальных параметров и секции локальных. Для краткости будем называть ее прообразом цепочки локального контекста. В данном случае считается, что локальных описаний нет, поэтому прообраз цепочки состоит из одного гнезда, стрелка из которого ведет к прообразу типа структуры формальных параметров, который на рисунке изображен узлом *тф*.

На рис. 4, *б* изображен результат генерации экземпляра процедуры из прообраза. В процессе генерации из прообраза глобальной контекстной цепочки создается экземпляр контекстной цепочки путем замены прообразов типов на экземпляры объектов, типы которых созданы из данных прообразов. Затем в соответствии с семантикой языка создается объект, представляющий собой объединение из прообраза процедуры и созданной контекстной цепочки.

На рис. 4, *в* изображен момент выполнения процедуры после ее вызова. Секция параметров представляет собой составной объект, расположенный в стеке активаций процедур. Он будет уничтожен при возврате из процедуры. В отличие от этого время жизни секции глобального контекста по семантике ДАДТ может быть не связано с последовательностью вызовов и возвратов из процедур. Поэтому в общем случае секция глобального контекста может находиться вне стека. В остальном структура секции формальных такая же, как у обычного составного объекта, в том числе есть связь с типом. Секцией активации процедуры назовем по традиции область стека, состоящую из секции формальных (и локальных, см. ниже) и слова возврата в вызванную процедуру (аналог запомненного состояния управления в ВМО). Логическая структура последнего близка к структуре экземпляра процедуры: здесь, как известно, запоминается связь с вызвавшей процедурой и контекстная цепочка места вызова.

Прообраз структуры. На рис. 5, *а* изображен прообраз структуры типа *m*, соответствующий генератору

конст *x* = **контекст огр** *e*1#*m*1, *e*2#*m*2 **ген** *m* **для**
(. . ., *p* = **проц** (. . .) (. . .))

при условии, что изображение типа имеет вид

тип m = структ (. . ., p)

Для связи с прообразом типа создаваемой структуры используется прообраз цепочки локального контекста (см. структурную модель процедуры). Чтобы пояснить смысл контекстной цепочки в прообразе структуры, в примере введен прообраз вложенной процедуры, контекстно подчиненной данной структуре и элементам ее контекстной цепочки.

Процесс генерации структуры из прообраза, результат которого изображен на рис. 5, *б*, можно в соответствии с семантикой ДАДТ представить следующим образом. Из прообраза контекстной цепочки, включая прообраз локальной цепочки, создается экземпляр, как при генерации процедурного объекта. Отличие заключается в том, что локальное гнездо заменяется на вновь созданный объект типа m с неопределенными элементами. Затем в контексте полученной цепочки последовательно выполняются выражения из правых частей описаний элементов в генераторе и объекты, полученные в результате выполнения, заменяют соответствующие неопределенные элементы новой структуры. В частности, в результате выполнения правой части описания p из прообраза процедуры будет создан процедурный объект, контекстной цепочкой которого становится цепочка, созданная в начале генерации, т. е. данная структура включается в контекст вложенной процедуры, как изображено на рис. 5, *б*.

На рис. 6, *а* изображен прообраз процедуры с локальными описаниями. Он представлен в соответствии с тем, как такие процедуры вводились в п. 5.4. В дальнейшем для краткости будем подобные процедуры изображать эквивалентной моделью, как представлено на рис. 6, *б*, объединяя при этом все вложенные прообразы в один пучок.

Еще одно эквивалентное преобразование состоит в том, что гнездо контекстной цепочки, соответствующее прообразу продолжения данной контекстной цепочки, можно заменять на прообраз контекстной цепочки того объекта, к которому приводит стрелка из этого гнезда. При этом прообраз цепочки сверху надстраивается несколькими гнездами.

Прообраз пакета обработки АТД. На рис. 7 изображена полная структурная модель процедуры генерации *генпак* пакета обработки АТД, рассмотренного в § 6. Опущены только несущественные для данного примера прообразы цепочек локального контекста процедур пакета. Важно отметить следующее. С учетом заданной в тексте *генпак* последовательности генерации объектов и используя приведенные выше правила генерации объектов из прообразов, по рис. 7 можно построить модель структуры памяти, образующейся в ре-

зультате выполнения процедуры *генпак*. Результат будет совпадать со структурой пакета, изображенной на рис. 2. При этом объекты пакета, отличные от структуры *в*, оказываются полностью недоступными пользователю пакета. Это обеспечивает два основных свойства АТД — доступность внутренних объектов пакета для операций

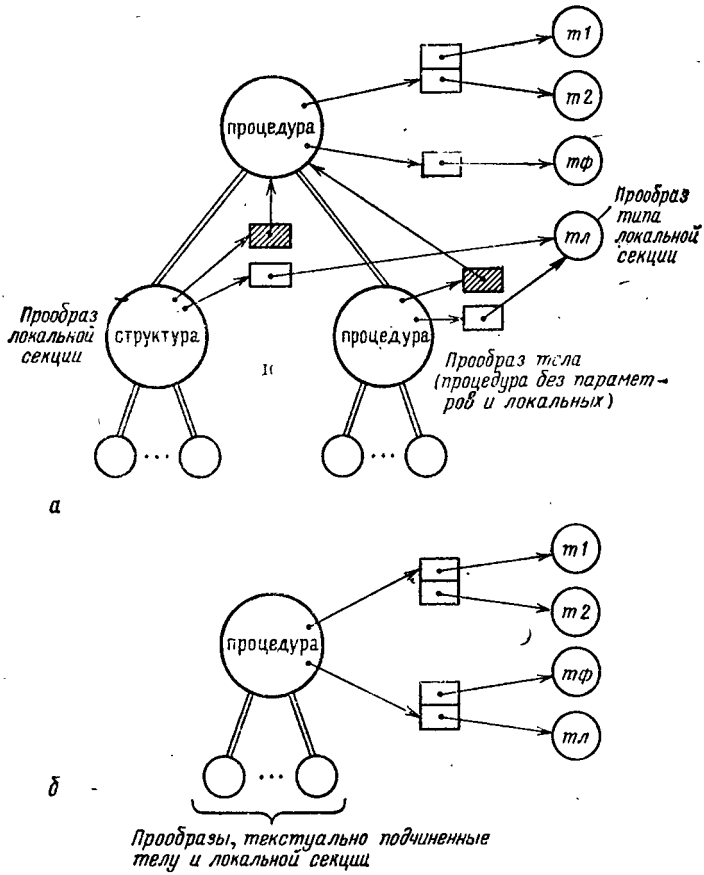


Рис. 6. Прообраз процедуры с локальной секцией; *а* — исходная модель; *б* — эквивалентная модель

обработки и ограничение доступа к объектам типа *m*, выдаваемым в контекст пользователя.

13.4. Поддержка компонентов СП. Одна из сложных проблем, возникающих при разработке СП, состоит в том, как обеспечить взаимодействие с пользователем в символьном виде в терминах исходной программы. Особенно это касается компонентов СП, обрабатываю-

щих выполняемую программу (простейший пример такого компонента — система аварийной выдачи). Известная сложность состоит в том, что в случае языков компиляционного типа в памяти нет символьной информации, касающейся обрабатываемых объектов.

Решение этой проблемы в системе Эльбрус основано на том, что структурная модель памяти выполняемой программы связана со

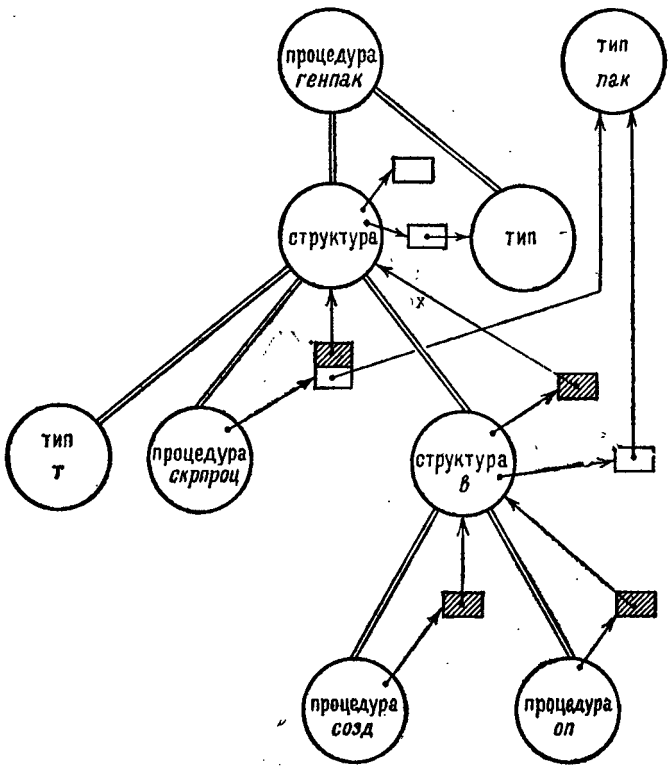


Рис. 7. Структурная модель процедуры генерации пакета обработки АТД

структурной моделью исходной программы так, что от объекта памяти можно прийти к узлу структуры программы, атрибутом которого является необходимая символьная информация. Это видно на рис. 4 и 5. Связь процедурного объекта с его прообразом запоминается в объекте в соответствии с семантикой ЯВУ. В случае составных объектов эта связь введена в тип составного объекта. Она приводит к прообразу типа, где хранится информация об идентификаторах элементов, о распределении памяти, о типах элементов и проч. Вопрос о

внутреннем представлении структурных моделей (в том числе символьной информации) и вопрос о реализации ссылки из экземпляра объекта, находящегося в памяти, на прообраз объекта мы относим к реализационным аспектам, которые рассмотрены в работах [19, 21, 30].

Резюмируя обсуждение структурных моделей, сформулируем реализованную в системе Эльбрус концепцию многообразия форм представления программы. В системе существует несколько стандартизованных форм внутреннего представления модели программы. Эти формы логически почти эквивалентны, а наличие разных представлений обусловлено различным характером обработки программы разными компонентами СП, аппаратурой и ОС. Основными статическими формами являются текстовая, кодовая, табличная. Кроме того, есть структура памяти выполняемой программы (СПВП), являющаяся совокупностью экземпляров объектов и экземпляров связей, созданных из их прообразов. Различные статические формы представления одного и того же объекта поставлены в соответствие одна другой, а элементы СПВП можно спроецировать на статическую форму представления исходной программы.

Процессор и ОС обрабатывают кодовую форму и СПВП; пользователь обрабатывает текстовую форму; СП, являющаяся посредником между системой и пользователем, обрабатывает как СПВП, так и статические формы, и проецирует СПВП на статические формы, чтобы вести взаимодействие с пользователем в терминах исходной программы. В этом с точки зрения разработки СП состоит существо организации процесса взаимодействия между системой и пользователем.

Для реализации этой концепции в системе Эльбрус разработаны и реализованы стандартные структуры файлов, соответствующих различным формам представления программы, механизм установления соответствия между различными статическими формами представления прообраза объекта, механизм отображения объекта на его статический прообраз и базовые операции обработки этих структур компонентами СП. В составе ОСПО Эльбрус реализован и эксплуатируется трансляторами и многоязыковыми инструментальными компонентами СП (система символьной отладки и аварийной выдачи, комплексатор программ и проч.) ряд пакетов, которые обеспечивают средства обработки структуры программы и СПВП, как структур абстрактного типа. Это пакеты ИНТЕРФОК, ИНТЕРКОД, ИНТЕРТЕКСТ и ИНТЕРПАМ [19, 29—33]. Использование пакетов не только облегчило разработку многоязыковой СП, взаимодействующей с пользователем в терминах исходной программы, но также упростило стыковку языков и перенос СП с одной модели семейства МВК Эльбрус на другую модель, имеющую отличия внутреннего представления стандартных файлов.

14. Некоторые тенденции развития архитектур и систем программирования

Рассмотрим ряд существующих подходов к усовершенствованию архитектурной поддержки СП для ЯВУ. Условно можно выделить два направления: во-первых, аппаратные реализации и, во-вторых, усовершенствование языков и программные реализации без изменения архитектуры машины.

14.1. Аппаратные реализации. Уже в период от 50-х годов до середины 70-х годов можно отметить эволюцию архитектур в сторону поддержки ЯВУ:

1. Такие черты традиционных систем, как виртуализация ресурсов, базированная адресация, программные соглашения об организации повторной входимости кода и процедурного механизма, аппаратная реализация магазинных операций — все это говорит о проникновении механизмов ЯВУ в архитектуру ОС и машин;

2. В 60-х годах был разработан отечественный проект ЭВМ Украина [67], в основу которого положена идея структурной интерпретации языка высокого уровня, нашедшая практическое воплощение в ЭВМ серии МИР. Это был, возможно, первый пример проекта ЭВМ, базирующегося на новых архитектурных принципах;

3. Серьезный шаг в сторону аппаратной реализации ЯВУ был сделан в машинах фирмы «Барроуз». Здесь на аппаратном уровне были реализованы: стек вызовов процедур и процедурный механизм, своеобразный механизм виртуальной памяти, тегированные данные.

Первые шаги в данном направлении, естественно, отличала привязка к конкретному ЯВУ. Например, ряд особенностей машин «Барроуз» показывает, что они в основном были ориентированы на языки, очень близко примыкающие к Алголу-60. Это подтверждается такими характерными признаками: отсутствие аппаратно реализованной контекстной защиты данных, специализация команд записи и считывания в зависимости от типа данных, жесткая привязка механизма адресации к стеку активаций процедур, ориентированная на языки с блочной структурой и затрудняющая реализацию механизма пакетов обработки АТД.

Из-за ограниченных возможностей Алгола-60 и для решения проблемы защиты системы от ошибок пользователя введен спектр из пяти чрезвычайно схожих алголоподобных языков [82]: для обычного пользователя, для системного программиста, для разработки системы телекоммуникаций, для разработки СУБД, для программирования заданий. Многообразие языков в данном случае возникает в основном не для решения задачи проблемной специализации ЯВУ, а как способ реализации защиты. Соответственно усложняется пользование системой: перед программистом стоит проблема выбора одного из нескольких похожих языков, причем при выборе

надо руководствоваться соображениями, связанными с проблемой защиты и/или эффективности, а не простоты программирования.

С другой стороны, надо подчеркнуть, что результаты, полученные в первых разработках архитектур, ориентированных на ЯВУ, послужили подтверждением перспективности данного направления и наметили проблемы его развития.

4. К концу 70-х годов рядом специалистов и производителей ЭВМ было признано, что для дальнейшего развития вычислительной техники нужно пойти на изменение архитектуры. Основные аспекты нового направления сформулированы ниже, следуя работам Г. Майерса и П. Денинга [54, 80]. Эти же аспекты отражены в Новой машине Неймана, разрабатываемой в проекте ЭВМ пятого поколения [68].

Надо отметить, что каждый из предложенных механизмов ранее уже был известен в вычислительной технике. Основная задача состоит в их сочленении и комплексном использовании. Чтобы определить место Эльбруса в рамках этого направления, отметим, что данная задача была решена и реализована в этой системе раньше, чем в рассмотренном ниже проекте, причем применительно к ЭВМ высокой производительности. Например, система команд МВК Эльбрус-1 была разработана в 1973 г., а окончательный вариант, сохранивший все принципиальные черты исходного,— в 1975—76 гг.

Рассматриваемое архитектурное направление предполагает аппаратную реализацию следующих механизмов:

— процедурный механизм;

— тегированная память. Существо механизма состоит в том, что в каждом слове памяти резервируется несколько разрядов, где хранится информация о типе данных. При выполнении операций контролируется правильность типов операндов. Наряду со стандартными типами должен существовать способ кодирования типов данных, определенных пользователем. В число стандартных типов целесообразно включать тип «пустое значение», что позволит диагностировать многие ошибки, вызванные использованием значения переменной до того, как было выполнено первое присваивание;

— контроль границ объекта. Например, должна аппаратно контролироваться величина индекса при обращении к элемеиту массива. В связи с этим должен существовать аппаратно-распознаваемый тип так называемых дескрипторов или указателей объектов, содержащих информацию о размере объекта;

— механизм виртуальной памяти для управления логическими сегментами переменной длины. В статье [80] сюда же включается механизм переключения контекста при вызове процедуры пакета обработки АТД. По нашему мнению, механизм АТД не требует специальной аппаратной поддержки. Как показано в § 2, переключение контекста можно отнести к процедурному механизму. Если при этом

необходимо считать в память сегмент данных пакета, то это действие выполняется механизмом виртуальной памяти;

— управление процессами. В основном сюда относят аппаратную реализацию техники семафоров и переключения процессов;

— механизм обработки ошибок, основанный на процедурном механизме с расширением, позволяющим после возникновения ошибки производить переход на код обработки ошибки. Отличие этого списка от перечня особенностей МВК Эльбрус состоит в основном в том, что несколько иначе систематизированы механизмы и не упомянуты многопроцессорность и высокое быстродействие. В остальном суммарные функциональные возможности совпадают, так как реализуются основные черты ЯВУ: типизация данных, включая механизм АТД и контроль типов за счет тегированной памяти; контекстная защита за счет сочетания процедурного механизма, тегированной памяти *) и контроля границ; динамичность управления памятью.

Подчеркнем, что подобная архитектура обладает той отличительной особенностью, что механизмы защиты и контроля будут «работать» всегда, в том числе при выполнении программ, использующих динамику типов, в частности, при выполнении системных программ.

Интересно отметить, что похожие архитектурные черты имеют некоторые 32-разрядные микропроцессоры 80-х годов, ориентированные на такие современные с точки зрения методов программирования языки, как Лисп и Smalltalk [83, 84]. Причем система команд реализована аппаратно, а не микропрограммно, т. е. основные из перечисленных механизмов не требуют значительных аппаратных затрат. Это говорит о том, что повышение быстродействия и степени интеграции позволяет и в классе микропроцессоров сочетать удобство программирования с эффективностью рабочих программ.

5. В проекте ЭВМ пятого поколения [68], базирующемся на достижениях технологии СБИС, предложен набор архитектур, одна из которых принадлежит рассматриваемому направлению. Предполагаются следующие архитектуры: машина логического вывода (ЛВ-машина), машина обработки функций (Ф-машина), машина обработки абстрактных типов данных в смысле процедурных языков, таких как Симула-67 и Клу (АТД-машина), машина с алгеброй отношений или машина реляционной базы данных (БД-машина), машина обработки потоков данных (ПД-машина), машина новой архитектуры Неймана (НН-машина). Назначение архитектур можно пояснить, соотнеся их с элементами функционирования систем Приз и Спора: (А) описание схемы БД или составление модели предметной области (МПО), причем специфи кации, связи с ППП хранятся в базе знаний; (Б) запрос к БД или постановка задачи на МПО; (В) планиро-

*) Для реализации контекстной защиты нужно по тегам распознавать процедурные объекты и дескрипторы составных объектов и контролировать правильность их обработки.

вание абстрактной программы на основе анализа запроса к БД или доказательство (с учетом семантики понятий МПО) существования решения задачи и извлечение алгоритма решения из доказательства; (Г) окончательный синтез программы и ее выполнение.

В качестве основной машины для работы с базой данных предполагается БД-машина. Для логического вывода на этапе В используется в основном ЛВ-машина. Возможно применение в качестве вспомогательных средств Ф-машины на этапах А, В (как машины более низкого уровня с функциями исполнительного механизма) и ЛВ- и Ф-машин при работе с базой знаний. Для операционной обработки на этапе Г применяются НН-машина (как универсальная машина операционной обработки), АТД-машина (формирование и обработка программ, использующих АТД) и ПД-машина (высокоскоростная параллельная обработка). Указанное подразделение архитектур может носить условный характер. Иногда требуется объединение функций различных машин. Поэтому окончательные архитектуры могут сочетать в себе различные возможности, взаимодополняющие друг друга.

Предполагается, что НН-машина по-прежнему ориентирована на операционный стиль программирования. Эта архитектура модернизируется путем удаления тех особенностей, которые, как отмечалось выше, были внесены из-за несовершенства элементной базы. К новым чертам архитектуры относят в основном такие: динамичность, механизм организации стека, механизм тегов (снабжающий данные информацией о типе) и полиморфные операции, механизм дескрипторов и АТД, управление большой виртуальной памятью, механизм защиты, многопроцессорность, высокое быстродействие. В проекте отмечается, что основная проблема состоит в комплексном объединении этих по отдельности известных механизмов в рамках одной вычислительной системы. Опыт разработки системы Эльбрус показывает, что решение этой проблемы естественным образом вытекает из аппаратной реализации основных механизмов ЯВУ. В этом случае оказывается, что удовлетворяется каждый из перечисленных пунктов *), т. е. появление БИС и особенно СБИС позволяет сочетать удобство программирования с высокой производительностью ЭВМ.

Особое место НН-машины в проекте связано с тем, что она выполняет комплексные функции: (а) как инструментальная ЭВМ для моделирования других архитектур и автоматизации проектирования машин, обеспечивающая в качестве важного средства преодоления сложности проекта высокий уровень и надежность программирования; в связи с этим высказывается мнение, что кроме архитектуры

*) Надо сделать оговорку, что многопроцессорность следует отнести к числу аппаратных средств повышения производительности и функциональной надежности комплекса.

Неймана в настоящее время нет другой архитектуры, предоставляющей такую же универсальность применения; (б) как одна из окончательных архитектур, предназначенная для обработки модулей ППП, написанных на процедурных языках, а также обеспечивающая совместимость с существующим багажом программ на ЯВУ; (в) в НН-машине отрабатывается реализация фундаментальных черт языков, имеющих отношение и к другим архитектурам, таких как типизация объектов и контроль обработки, механизм процедур и методы защиты данных. Наконец, как видно из списка черт НН-машины, возможности архитектур Ф-машины (в варианте Лиспа), АТД- и НН-машины пересекаются, и они, в заключение, могут образовать единую архитектуру.

Таким образом, с точки зрения методов программирования следующего поколения процедурные языки и соответствующие им архитектуры имеют одновременно и практическое и самостоятельное теоретическое значение.

Резюмируя, отметим следующее обстоятельство. Как видно из анализа архитектурного направления, связанного с реализацией механизмов ЯВУ, оно основывается на том соображении, что для преодоления сложности разработки программных систем, в частности, для увеличения надежности программирования нужна аппаратная реализация языковых механизмов контроля и защиты.

14.2. Программные реализации. Примеры усовершенствования средств программирования путем создания многоуровневых языков, а также путем изготовления программных надстроек над машиной и ОС были рассмотрены в § 12 и в [102, 103].

В целом надо отметить, что программно-языковое надстраивание существующих систем остается практически ценным приемом «сглаживания» архитектурных рассогласований между системой и ЯВУ. Теоретическое значение этих работ состоит в том, что в них в сущности отработаны элементы языкоориентированных архитектур. В качестве примеров отечественных разработок, принадлежащих данному направлению, можно назвать: один из первых в мире динамических загрузчиков системы ИС-2 для М-20 [69]; язык Алмо [70], отличительная особенность которого состоит в том, что эта работа была, пожалуй, одной из первых, где цель упрощения разработки СП и универсализации ее компонентов достигалась путем введения промежуточного языка достаточно высокого уровня; язык АБВ [65], обеспечивающий компактный и ортогональный набор машинно-независимых примитивов построения семантических моделей механизмов ЯВУ для реализации языков; языки Ярмо [71] и Астра [72], широко используемые для создания программного обеспечения БЭСМ-6; система Альфа [73], в которой наряду с компилятором имеются подсистемы управления архивом, подготовкой данных, редактированием, символьной отладкой, а также язык управления систе-

мой. Сюда же надо отнести отечественные работы по многоязыковым системам: система Бета [74, 75], развивающая вслед за системой Альфа идеи глобальной оптимизации в рамках многоязыковой системы; Р-технология, работы ЛГУ, ИТА АН СССР, ИК АН ЭССР, НФ ИТМиВТ, ИК АН УССР, ИМ АН МССР (см. подробный обзор в кн. [76]).

15. Выводы

Сформулируем основные выводы, касающиеся разработок языка Эль-76 и средств системной поддержки инструментальных компонентов СП для ЯВУ.

15.1. Базовые механизмы языков. В этой главе были выделены базовые механизмы ЯВУ, затем введены некоторые обобщения, и в результате получены элементы гипотетического языка, имеющего расширенные возможности в части динамики управления объектами и типами, включая объекты абстрактных типов, и в части обработки архива. К основным характеристикам языка можно отнести следующие:

1. Типизация и структурирование объектов. Типы разбиты на классы скалярных типов, имен, составных объектов (структур и массивов) и процедурных объектов. Для каждого класса введены операции с динамическим контролем типов.

2. Реализация принципа универсальной применимости базовых средств обработки объектов, в соответствии с которым значением переменной, параметром и выдачей процедуры, элементом составного объекта может быть объект любого типа. Для всех объектов используется механизм именования в распределенном контексте и средства динамического управления временем жизни.

3. Механизм распределенного контекста, который наряду с известными чертами обладает следующими особенностями:

— контекст рассматривается как совокупность структур. Это позволяет ввести процедуры, контекстно подчиненные структурам;

— контекстная подчиненность процедуры может быть задана в программе явным образом. Это дает возможность ослабить связь между контекстной и структурной подчиненностью и, как следствие, позволяет описывать процедуры, выдающие в качестве результата вложенные процедуры (возможно, входящие в состав выдаваемой структуры), и описывать на языке динамическое управление модульными конфигурациями;

— механизм контроля контекстной известности типов используется как средство, позволяющее полностью скрыть структуру объекта, выдаваемого в контекст пользователя;

— средства управления внешними объектами, опирающиеся на общий механизм управления контекстом и правила определения вре-

мени жизни объектов. Это дает возможность описывать на языке полный цикл обработки (т. е. создание объекта, его использование и уничтожение) архива файлов, программных модулей, в том числе определений АТД. Причем на эти процессы обработки распространяются основные механизмы ЯВУ; контекстная защита, контроль типов.

Такого набора элементов языка оказывается достаточно, чтобы, не вводя специальных конструкций, можно было программировать определение АТД (т. е. описание набора операций над объектами абстрактных типов и, возможно, нескольких вариантов реализации операций и структур данных), описывать раздельно транслируемые модули, их сочленение в единую программу и проч.

4. Механизм модульной обработки ошибок, как расширение набора структур управления последовательностью действий.

Кроме того, в этой главе показано, что статический контроль типов можно ввести, дав средство описания некоторых статических свойств алгоритмов, а также определен набор этих свойств.

Резюмируя обсуждение элементов гипотетического языка, выделим механизмы, составляющие основу средств поддержки модульности и динамичности в ЯВУ: механизм распределенного контекста и вытекающая из него контекстная защита; механизмы типизации и структурирования и обусловленные ими семантический контроль вычислений и средства ограничения прав доступа к объектам; механизмы модульного динамического управления объектами и типами. Эти механизмы считались базовыми при создании конкретного языка программирования Эль-76.

15.2. Некоторые тенденции развития архитектур. Анализ задачи повышения эффективности программирования систем (в широком смысле) путем полного перевода всего профессионального программирования на ЯВУ в сочетании с применением развитых систем поддержки программирования (СПП) показывает следующее:

1. Ограниченные функциональные возможности распространенных ЯВУ и недостаточно эффективная реализация являются причиной того, что в ряде областей традиционно используются ассемблерные средства программирования и на практике зачастую приходится использовать одновременно три языка, отличающихся по выразительным средствам, а именно Ассемблер, язык управления операционной системой и ЯВУ. В целом это усложняет программирование и снижает надежность программ.

2. Для реализации развитых СПП требуется предварительное изготовление значительной программной надстройки над базовой системой, что усложняет создание СПП. В результате разработка и широкое применение подобных СПП являются на сегодняшний день скорее исключением, нежели правилом в практике программирования.

3. Основной причиной, затрудняющей разработку ЯВУ, обладающих одновременно свойствами надежности, универсальной применимости и эффективности, а также разработку СПП для этих языков, является рассогласование между базовыми механизмами ЯВУ и СПП, с одной стороны, архитектурой ЭВМ и ОС, с другой стороны. Характерными признаками рассогласования являются следующие:

— структурирование и типизация объектов и соответствующий этим чертам семантический контроль обработки в ЯВУ, в отличие от хранения неструктурированной информации и неконтролируемых вычислений в ЭВМ;

— механизм распределенного контекста в ЯВУ, обеспечивающий частичное или полное ограничение доступа к объектам, в отличие от механизма адресации общей неструктурированной памяти в ЭВМ и методов именования архивных объектов, применяемых в ОС;

— недостаточная поддержка со стороны аппаратуры и ОС для организации присущего ЯВУ динамического управления временем жизни объектов, включая архивные объекты;

— структурирование программных конфигураций архивных объектов в СПП, в отличие от плоского архива, предоставляемого традиционными ОС.

4. Анализ основных направлений развития средств программирования показывает, что в области разработки архитектур ЭВМ имеет место тенденция к устранению семантического разрыва между архитектурой и ЯВУ. В программных реализациях этот разрыв устраняется путем изготовления программной надстройки над базовой системой (аппаратурой и ОС). Вместе с тем ориентация архитектуры на конкретный ЯВУ не решает задачи, так как сохраняются методы неконтролируемого программирования.

15.3. Выбор базового языка программирования. Рассмотрим некоторые варианты выбора базового языка программирования для новой вычислительной системы, соотнося вопрос выбора языка с задачей разработки архитектуры.

1. Разработка новой архитектуры, ориентированной на существующий ЯВУ. Опыт разработок машин, ориентированных на Алгол (см. п. 14.1), показывает, что такой подход не решает проблему. Причина состоит в том, что опосредованно через язык (точнее, через ограничения, наложенные на язык из соображений эффективности реализации) предыдущие архитектуры влияют на новую архитектуру. Как следствие, в системе приходится сохранять методы неконтролируемого программирования, в том числе применяемые в прикладных программах.

2. Разработка машинно-ориентированного языка на базе существующей архитектуры. Этот вариант, не подразумевающий разработку новой архитектуры, включен сюда для полноты перебора. В § 12 было показано, что методы неконтролируемого программиро-

вания по-прежнему остаются. Причина в том, что требование приспособления языка к наперед заданной архитектуре приводит к необходимости вносить в язык различные особенности, снижающие уровень контроля.

3. Иногда предлагается вариант использования машинно-ориентированного языка, разработанного для традиционных архитектур, в качестве базового языка программирования на новой архитектуре, обеспечивающей контроль и защиту. Такой подход может потребовать такой модернизации языка, которая принципиально изменит его лицо. Например, в таких языках, как Блисс, есть средства, позволяющие трактовать произвольное число как адрес объекта. Эта особенность, являющаяся источником нарушения защиты, в сущности означает, что память выполняемой программы рассматривается как неструктурированное образование. Действительно, это имеет место в машинах традиционной архитектуры (линейная неструктурированная память), для которой язык первоначально создавался и в среде которой данное языковое решение было оправданным.

Однако в случае архитектуры, ориентированной на ЯВУ, дело обстоит иначе. Здесь реализована одна из основных черт ЯВУ — структуризация памяти, т. е. разбиение ее на логические объекты с контролем границ объектов в процессе его обработки (и в этом заключен один из основных элементов языковой защиты). Чтобы использовать в языке эту новую черту архитектуры, надо убрать конструкции, связанные с неструктурированной памятью, и наоборот, ввести новые конструкции, присущие обычному процедурному ЯВУ. Далее, в машине с аппаратной реализацией процедурного механизма (как элемента обеспечения защиты) и динамическим распределением ресурсов можно отказаться от языковых средств базирования, сохранения регистров, описания размера стека. Это упростит соответствующий раздел результирующего языка и соответственно его применение.

Таким образом, можно заключить, что необходимость принципиальной модернизации при переходе к новой архитектуре делает нецелесообразным применение машинно-ориентированного ЯВУ, разработанного для другого класса архитектур. Это, конечно, не означает, что такой язык нельзя моделировать на новой архитектуре с целью обеспечения переноса программ.

Проанализируем вопрос применения языка Си в качестве базового языка программирования для новой архитектуры. Как отмечалось выше, в языке сохранены методы неконтролируемого программирования. Эта черта характерна (и вполне обоснованна) для языков подобного класса, так как они должны давать возможность программировать сложные системы управления динамической обстановкой, для которых концепция статического контроля недостаточно хорошо приспособлена. Можно сказать, что в случае традици-

онных архитектур нет иной альтернативы построения таких языков.

В условиях новой аппаратно-программной разработки возможен иной подход к созданию языка программирования систем. Если в качестве основы выбрать метод динамического управления типами, то контроль и защита обеспечиваются и для систем указанного выше класса. Но такой метод требует создания нового языка, а для его эффективной реализации нужна архитектура с тегированной памятью. Одновременно эта архитектура уже на аппаратном уровне обеспечивает защиту и типизацию данных. В рамках теговой архитектуры, разумеется, можно реализовать статический язык и использовать ее новые качества, чтобы усилить контроль в языке. Но использование его в качестве базового, как видно, нецелесообразно, так как это скроет от программиста новые возможности архитектуры, а именно средства динамического управления типами.

4. Последний вариант — это совместная разработка языка и новой архитектуры, опирающаяся на совокупность фундаментальных механизмов ЯВУ. В результате устранения семантического разрыва между архитектурой и ЯВУ появляются дополнительные возможности. Во-первых, применение ЯВУ не приводит к потере номинальной производительности машины. Во-вторых, повышается надежность программ, так как механизмы семантического контроля и защиты становятся стандартом для реализации и разработки ЯВУ, поскольку эти механизмы реализованы аппаратно.

В-третьих, повышение уровня архитектуры упрощает разработку СПП для ЯВУ тем, что необходимая для СПП программная надстройка оказывается уже реализованной в аппаратуре и ОС. Еще одно положительное следствие этого подхода становится понятно, если учесть, что будет продолжаться программирование на существующих ЯВУ, будут появляться многоязыковые программы, использующие накопленный программный багаж, например, пакеты, написанные на разных языках. Здесь возникает проблема стыковки языков и разработки многоязыковых СПП. Решение этой проблемы также упрощается, если ввести достаточно высокий уровень архитектурных стандартов на механизмы поддержки языков и СПП. В этом случае различные языки получают готовые системные механизмы поддержки, и причин для разночтений в реализации схожих объектов становится существенно меньше, чем в традиционном случае. Следовательно, упрощаются стыковка программ, написанных на разных языках, и разработка многоязыковых СПП.

15.4. Совместная разработка языка и системы. Основные положения совместной разработки языка Эль-76 и архитектуры системы Эльбрус сводятся к следующему. Чтобы обеспечить достаточную универсальность с точки зрения реализации языков программирования, архитектура системы разрабатывалась, опираясь не на

существующий ЯВУ или какой-либо конкретный язык, а на совокупность базовых механизмов ЯВУ. Совместно с архитектурой создавался новый ЯВУ Эль-76, использование в котором особенностей этой архитектуры позволило избежать традиционных ограничений ЯВУ и получить универсально применимый ЯВУ для программирования систем. Этот язык является эффективным в том смысле, что адекватно отображая архитектуру, позволяет полностью использовать возможности и номинальную производительность машины. Это первое следствие данного подхода.

Второе следствие состоит в том, что повышается эффективность реализации традиционных ЯВУ. Третье следствие системной реализации механизмов ЯВУ состоит в том, что на уровне системы сохраняется информация о структурных свойствах исходной программы на ЯВУ, т. е. ликвидируется соответствующий недостаток традиционных архитектур, затрудняющий разработку СПП, общающихся с пользователем в терминах исходной программы на ЯВУ. В связи с этим в качестве неотъемлемой части архитектуры разработаны средства технологической поддержки для создания СПП. Такой подход хорошо согласуется с общей идеей устранения семантического разрыва между архитектурой и ЯВУ. В практическом отношении это позволяет упростить разработку СПП для ЯВУ, в том числе — многоязыковых СПП, и тем самым поддержать реализацию второго из указанных в § 1 направлений полного перевода программирования на ЯВУ.

В целом перевод программирования на ЯВУ в системе Эльбрус обеспечивается сочетанием следующих элементов:

— применение языка Эль-76 для программирования систем с повышенными требованиями к языку программирования (эффективность, доступ ко всем возможностям системы, динамичность, надежность);

— эффективное применение существующих ЯВУ без отключения семантического контроля;

— разработка программ на ЯВУ, использующих готовые модули, написанные на других ЯВУ;

— применение, в первую очередь для отладки, СП, общающихся с пользователем исключительно в терминах ЯВУ.

ГЛАВА 2

БАЗОВЫЕ КОНСТРУКЦИИ

1. Основные понятия

Текст программы содержит описание алгоритма обработки объектов. В процессе выполнения алгоритма могут предприниматься следующие основные действия: арифметические, логические и другие стандартные, т. е. определенные в описании языка операции над объектами стандартных типов; присваивание, выполняющее замену текущего значения переменной; создание и уничтожение объектов стандартных и определяемых типов, а также вызов процедуры, реализующей нестандартные, т. е. определенные в программе операции над объектами стандартных и программно определяемых типов. К числу основных действий следует отнести также операции над внешними объектами, в частности, ввод/вывод и поиск в архиве.

Структурирование текста и действий программы производится с помощью процедур, закрытых предложений и модулей. Процедура может иметь параметры, обеспечивающие ее информационную связь с другими частями программы. При вызове процедуры происходит замена формальных параметров на фактические, после чего управление передается вызываемой процедуре.

Модуль обычно используется как средство организации совокупности (пакета) процедур, реализующих набор операций над объектами одного или нескольких стандартных или программно определяемых типов.

В число закрытых предложений входят: условное и выбирающее предложение, цикл и структурное предложение. Первые два выбирают одну из заключенных внутри них альтернатив и передают ей управление. Каждая из альтернатив является последовательным предложением, задающим прямую (текстуальную) последовательность действий. Цикл организует повторное выполнение последовательного предложения. Структурное предложение и связанный с ним механизм ситуаций позволяет при заданных обстоятельствах прекращать выполнение закрытого предложения еще до того, как

будет достигнут текстуальный конец последнего. Это средство используется и для обработки исключительных обстоятельств (например, ошибок), и для описания обычных алгоритмов. Например, структурное предложение может быть использовано для программирования окончания цикла.

1.1. Объекты, константы и переменные. Объект характеризуется типом. Типы разбиваются на четыре класса: атомарные типы, типы имен, составные типы и процедурные типы.

Константы. Константой называется совокупность, состоящая из объекта и обозначения, с помощью которого объект упоминается в программе. Обозначаемый объект называется значением константы. В случае константы связь между обозначением и обозначаемым объектом неразрывна, т. е. объект, находящийся на конце связи, не может быть заменен на другой объект. Базовая форма описания константы имеет вид:

конст ⟨идентификатор⟩ = ⟨выражение⟩

Такое описание вводит константу, обозначением которой является ⟨идентификатор⟩, заданный в левой части описания, а значением — объект, являющийся результатом выполнения ⟨выражения⟩, находящегося в правой части описания.

Существуют константы статического класса и константы динамического класса. Разница между ними в том, что для констант первого класса память не отводится (они содержатся в коде программы), а для констант второго класса — отводится.

В языке введены препроцессорные константы: описатели полей и текстовые макросы. Особенность этих констант состоит в том, что они используются только во время трансляции, как макросредства.

Константы встроенные и отдельно размещенные. С точки зрения реализации языка, описание константы с ⟨выражением⟩, являющимся генератором объекта, т. е. конструкцией, создающей новый объект, имеет особенность, которая состоит в том, что объект-значение константы создается в момент выполнения описания. Такая форма описания допускает ряд оптимизаций как в части управления памятью, так и в части эффективности доступа к объекту. Чтобы в программах можно было явным образом управлять этой возможностью, используя при этом сокращенную форму записи, введены так называемые встроенные константы. Описание такой константы, безусловно, совмещено с генерацией объекта. Константы, не являющиеся встроенными, называются отдельно размещенными.

Переменные. Объект типа имя обладает свойством именовать другие объекты. Связь между именем и именуемым объектом переменная, т. е. именуемый объект, находящийся на конце связи, может заме-

няться на другой объект, который в результате этого действия становится объектом, именуемым данным именем. Это действие производится с помощью стандартной операции присваивания, определенной над именами. Второй стандартной операцией является операция «разыменование», выдающая именуемый объект. Совокупность, состоящая из имени и именуемого объекта, называется переменной. Переменная вводится специальной формой описания, называемой описанием переменной. Именуемый объект называется значением переменной.

1.2. Типы объектов. В языке есть один стандартный атомарный тип, называемый логическим типом. Существуют два объекта логического типа — **истина** и **ложь**. Эти объекты являются неделимыми.

Составные типы. Объект составного типа состоит из элементов, являющихся константами или переменными; тип константы или переменной может быть динамическим (см. ниже определение понятия типа константы и переменной). В языке существуют средства для конструирования следующих составных типов:

— тип массива. Массив состоит из элементов одинакового (возможно, динамического) типа. Тип массива определяет число измерений и тип элемента. Тип массива является параметризованным типом. Параметрами типа, называемыми параметрами длины, являются длины измерений. При создании массива данного типа задаются фактические значения параметров, которые определяют конкретные длины измерений создаваемого объекта. Каждому элементу n-мерного массива соответствует n индексов, однозначно идентифицирующих этот элемент;

— тип области. Область состоит из элементов одинакового типа, но, возможно, различного размера. Тип области является параметризованным типом. Параметр типа называется параметром длины. Тип параметра длины определяет максимально допустимое количество элементов в областях данного типа. В отличие от массива, все элементы которого создаются при создании массива, элементы области можно создавать и ликвидировать по отдельности. При создании области задается фактическое значение параметра, определяющее максимально возможное число элементов в этой области. Тип элемента области может быть параметризованным типом. При создании элемента задаются фактические параметры типа элемента;

— тип структуры. Структура состоит из элементов, типы которых могут быть различными. Тип структуры может быть параметризованным типом, т. е. может иметь формальные параметры. Параметр структуры может использоваться либо для того, чтобы задавать длину измерения элемента этой структуры, имеющего тип массива, либо для выбора типа элемента, заданного с помощью объединенного типа (см. ниже), либо для транзитной передачи в ка-

честве фактического параметра типа элемента. Типом последнего в свою очередь является тип структуры;

— тип модулей. Модуль состоит из элементов, возможно, различных типов. Основные отличия модуля от структуры состоят в следующем: описание типа модуля может содержать выражения (в том числе процедуры), выполняемые в контексте идентификаторов элементов модуля; модуль в общем случае состоит из двух составных объектов: из интерфейса модуля и его скрытой части. Элементы интерфейса, в том числе интерфейсные процедуры, доступны пользователю модуля; элементы скрытой части модуля недоступны пользователю, но их можно использовать в интерфейсных процедурах; время жизни модуля не ограничено временем выполнения его интерфейсных и скрытых процедур;

— тип процедуры определяет совокупность и типы формальных параметров процедуры, а также в случае процедуры-функции — тип результата.

1.3. Множества и типы. Перечисленные выше типы называются первичными типами. Каждый объект имеет тот или иной первичный тип. Объекты одного и того же первичного типа образуют первичное множество объектов этого типа.

Подтип первичного типа. Существует несколько способов выделения подмножества первичного множества:

— сокращение диапазона типа целого. Этот способ выделяет подмножество типа целого от 0 до заданной верхней границы;

— связывание параметров составного типа. Этот способ выделяет из первичного множества (или его подмножества) все объекты, имеющие заданные значения параметров. Совокупность, состоящая из типа исходного множества и заданных значений параметров, называется типом выделенного таким способом подмножества. Тип этого подмножества называется подтипом типа исходного множества;

— ограничение областью. Этот способ выделяет из первичного множества или его подмножества все объекты, являющиеся элементами заданной области. Совокупность, состоящая из заданной области и типа исходного множества, называется типом выделенного таким способом подмножества. Тип этого подмножества является подтипом типа исходного множества.

Как отмечалось выше, каждый объект, в том числе входящий в подмножество, фактически является объектом первичного типа. Однако если необходимо подчеркнуть принадлежность объекта подмножеству, тип которого является подтипом некоторого типа m, говорится, что тип объекта является подтипом типа m. Аналогично, если необходимо подчеркнуть принадлежность типа объекта объединенному множеству (см. ниже), говорится, что объект имеет

объединенный тип, являющийся типом этого объединенного множества.

Объединенный тип. Объединенным множеством называется множество, состоящее из непересекающихся множеств. Типом объединенного множества (или объединенным типом) называется совокупность типов составляющих его множеств. Составляющие типы являются подтипами объединенного типа. Подтип объединенного типа может одновременно являться подтипом другого типа.

Динамический тип. Динамическим типом называется тип объединенного множества, составленного из всех непересекающихся множеств.

Объект типа тип. Все перечисленные выше типы сами являются объектами одного и того же предопределенного типа *тип*.

Тип константы и переменной. Тип множества, объекты которого могут являться значением константы (переменной), называется типом константы (переменной). Тип константы и переменной может быть первичным типом или его подтипом или объединенным типом, в том числе — динамическим типом.

1.4. Форматы. При определении объектов некоторых стандартных типов, например — чисел, используется термин «формат» объекта. Форматом называется количество разрядов, занимаемое объектом. Например, вещественное число может занимать 32, 64 или 128 разрядов (от формата числа зависит точность его представления, а также максимально и минимально возможная абсолютная величина). Кроме того, формат является характеристикой переменной и константы, введенной с помощью предварительного описания. Он определяет размер отводимой для них области памяти.

Стандартными форматами являются следующие: бит — 1 разряд (ф1), тетрада — 4 разряда (ф4), байтовый или литерный формат — 8 разрядов (ф8), половина слова — 32 разряда (ф32), слово — 64 разряда (ф64), два слова — 128 разрядов (ф128). Первые три формата называются строчными, остальные — простыми.

Переменными (константами) простого формата называются переменные (константы) динамического типа, имеющие простой формат, или переменные (константы) стандартных числовых типов и типа *наб*, причем последняя имеет формат ф64. Переменными (константами) строчных форматов называются переменные (константы), имеющие строчный формат, или переменные (константы) стандартных типов *лог* (формат ф1), *циф* (формат ф4) и *лит* (формат ф8).

Описание переменной динамического типа задается с помощью служебных слов **перем** (формат ф64) или явного указания формата **ф128, ф64, ф32, ф8, ф4, ф1.** Последние три формата возможны только в случае массива переменных. Предварительное описание константы динамического типа, имеющей формат ф64, задается с помощью служебного слова **конст.**

1.5. Обозначения и конструкции. Текст программы строится из конструкций — описаний и предложений.

О п и с а н и е вводит константу или переменную и связывает с ней обозначение (идентификатор).

П р е д л о ж е н и е может обозначать:

— объект, явно заданный своим изображением, например: 3.14 (число), **истина** (логическое значение);

— действия над объектами, например: $x+y$. Ряд предложений, обозначающих действия, обладает тем свойством, что в результате их выполнения получается некоторый объект. Например, результат выполнения $x+y$ есть некоторое число. Будем говорить, что такие предложения выдают значение. Этим значением является результирующий объект. Его для краткости будем называть значением предложения;

— переменную или константу, например: x — идентификатор переменной или константы, $a[i]$ — обозначение элемента массива (последний является переменной или константой, см. п. 6.2). В состав подобных обозначений могут входить обозначения для действий, например: $i+1$ в предложении $a[i+1]$. При этом предполагается, что сначала выполняются действия, а затем определяется обозначаемая переменная или константа;

— последовательность, в которой выполняются действия, например:

если q **то** $x := x+y$ **иначе** $a[i] := 3.14$ **все.**

Тип предложения. В описании языка при определении семантики предложений, выдающих значение, указывается тип множества, объекты которого могут являться значением этого предложения. Этот тип называется типом предложения. Следует различать тип предложения и тип значения предложения, поскольку второй тип может быть подтипом первого типа.

Операндом операции называется предложение, значение которого обрабатывается данной операцией. В ряде случаев для сокращения изложения не указан тип операнда операции, а указан только тип значения операнда. В этих случаях подразумевается, что тип операнда либо может быть динамическим, а значением операнда является один из указанных стандартных типов, либо тип операнда и тип значения статически известны и совпадают.

1.6. Динамическое распределение памяти, локализации. Память, необходимая для переменных и констант динамического класса, являющихся элементами составного объекта, отводится динамически при создании объекта. Автоматическое освобождение памяти происходит по истечении времени жизни объекта.

Объекты бывают о п е р а т и в н ы е и в н е ш н и е. Время жизни оперативного объекта связано со временем выполнения тех

или иных предложений программы. Внешними объектами являются объекты, хранящиеся в архиве (например — файлы). Время жизни этих объектов может быть не связано со временем выполнения программ, которые их создают и используют.

Время жизни объекта определяется его локализацией. Оперативный объект может быть локализован в модуле или в задаче. Время жизни объекта, локализованного в модуле, совпадает со временем жизни данного модуля. Объект, локализованный в задаче, существует до конца задачи. Такой объект называется г л о б а л ь н ы м. Для процедурного объекта также вводится понятие времени жизни (см. далее п. 11.1).

Б л о к о м называется процедура, закрытое предложение, начинающееся с описаний, и последовательное предложение, начинающееся с описаний. Совокупность описаний, с которых начинается блок, рассматривается как особый способ определения модульного объекта, который существует до конца выполнения данного блока.

В языке существуют средства явного управления локализацией объектов. Если время жизни явно не задано, то по умолчанию выбирается локализация в ближайшем текстуально охватывающем модуле или модуле, составленном из описаний ближайшего текстуально охватывающего блока. Такой объект называется л о к а л ь н ы м.

Внешний объект может быть локализован в модуле или в задаче, или может быть постоянным. В первых двух случаях время жизни внешнего объекта определяется, как указано выше. Объект называется постоянным, если есть другие внешние объекты, содержащие ссылки на него. Постоянный объект существует до тех пор, пока существуют подобные ссылки. Для объектов всех типов есть средства явного уничтожения.

1.7. Статус привилегированности. Процедура может быть привилегированной и непривилегированной. В непривилегированных процедурах запрещены привилегированные действия: изменение поля адресного объекта (указатель, метка) *), изменение типа неадресного объекта на какой-либо адресный тип. При нарушении этих ограничений возникает ситуация *привдействия*.

Статус процедуры определяется статусом программы. Привилегированная программа может содержать процедуры обеих разновидностей, а непривилегированная — только непривилегированные процедуры. Аналогичным образом статус программы зависит от статуса пользователя. Статус пользователя назначается при регистрации его в системе. Необходимо отметить, что динамически вызываемые процедуры и программы сохраняют свой собственный ста-

*) Введение статуса привилегированности — это технический прием, с помощью которого реализовано отделение системного стандартного контекста операционной системы от стандартного контекста программ пользователей.

тус. Например, непривилегированный пользователь может вызывать привилегированные процедуры операционной системы, находящиеся в доступном ему контексте. При этом последние сохраняют право выполнять привилегированные действия. ■

1.8. Дополнительные возможности. В язык встроены групповые операции обработки битовых, цифровых и литерных векторов. В число этих операций входит пересылка векторов, сравнение, поиск определенного элемента, операции перевода чисел (двоичное ↔ шестнадцатеричное ↔ литерное). Для векторов, состоящих из элементов простых форматов, введены только пересылка и поиск.

Определение текста и подстановка текста рассматриваются как простые макросредства и предназначены для сокращения и структурирования текста программ.

Конструкции, связанные с преобразованием формы массива, позволяют изменить логическую структуру многомерного массива, обрабатывать сечение массива как самостоятельный объект и проч. При этом элементы исходного массива не копируются в какой-либо другой массив, создаваемый явно или неявно.

Кроме перечисленного, в языке существуют средства обработки внутреннего представления объектов.

1.9. Метод описания синтаксиса языка. Синтаксис языка описан с помощью порождающих правил для понятий. Каждое правило имеет следующий вид:

левая часть ::= правая часть

Синтаксические понятия заключаются в угловые скобки. Альтернативы в правой части разделяются вертикальной чертой.

Если в некоторой альтернативе часть конструкции может отсутствовать, то она заключается в фигурные скобки. Три точки после синтаксического понятия означают, что оно может повторяться один или более раз. Аналогично три точки после части конструкции, заключенной в фигурные скобки, означают, что она может повторяться нуль или более раз.

Если в правой части порождающего правила некоторое понятие имеет вид ⟨список X⟩, где X — синтаксическое понятие, то тем самым неявно задано правило:

⟨список X⟩ ::= ⟨X⟩ {, ⟨X⟩}...

Если в описании семантики употребляется синтаксическое понятие, то оно также заключается в угловые скобки.

Чтобы не перегружать синтаксис языка, некоторые уточнения синтаксического характера вынесены в семантику.

2. Базовые обозначения

2.1. Базовые символы. Конструкции языка представляются с помощью набора базовых символов, являющегося подмножеством кода ДКОИ-8. В этот набор входят:

— русские и латинские буквы верхнего регистра

— цифры от 0 до 9

— специальные символы [. < (+ @ ^] ▢ *) & ; : ' # / | > , % ! " — = _

— пробел.

2.2. Лексические элементы. Лексический элемент языка — это ⟨идентификатор⟩, ⟨число⟩, ⟨поэлементное представление⟩, подчеркнутый идентификатор или разделитель. Лексические элементы строятся из ⟨букв⟩, ⟨цифр⟩, ⟨двоичных цифр⟩, ⟨восьмеричных цифр⟩, ⟨шестнадцатеричных цифр⟩ и ⟨литер⟩.

⟨буква⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | У | Ю | Б | Ц | Д | Ф | Г | Й | Л | Я | Ч | Ж | Ы | Ь | З | Ш | Э | Ъ

⟨цифра⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨двоичная цифра⟩ ::= 0 | 1

⟨восьмеричная цифра⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

⟨шестнадцатеричная цифра⟩ ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F

⟨литера⟩ ::= ⟨буква⟩ | ⟨цифра⟩ | ⟨вертикальная черта⟩ | ⟨кавычка⟩ | [| . | < | (| + | @ | ^ |] | ▢ | * | & |) | ; | : | ' | # | , | % | > | ! | — | = | _ | ⟨пробел⟩ | /

В тексте книги ⟨буквы⟩ языка набраны курсивом. Следует учитывать, что буквы И и U, П и N, Т (русск.) и M (лат.) имеют разные начертания в верхнем регистре, но одинаковые в нижнем. В дальнейшем, где не оговорено особо, подразумевается, что *и*, *п* и *т* представляют соответствующие буквы русского алфавита.

⟨Вертикальная черта⟩ — это символ «|». ⟨Кавычка⟩, встречающаяся внутри ⟨поэлементного представления⟩ (т. е. не являющаяся ограничителем), обозначается двумя кавычками (" ").

Числовая величина ⟨литеры⟩ задается ее внутренним двоичным представлением. В операциях сравнения считается, что литеры упорядочены по их числовым величинам.

Идентификаторы используются для обозначения переменных и констант, вводимых с помощью описаний.

⟨идентификатор⟩ ::= ⟨буква⟩ { ⟨буква или цифра⟩} ...
⟨буква или цифра⟩ ::= ⟨буква⟩ | ⟨цифра⟩

Ч и с л а. ⟨Целое⟩ и ⟨вещественное⟩ — это десятичные числа в их обычном понимании. ⟨Порядок⟩ представляет собой степень числа 10.

⟨целое⟩ ::= ⟨цифра⟩ ...
⟨вещественное⟩ ::= ⟨целое⟩ . ⟨целое⟩ { ⟨порядок⟩ } | ⟨целое⟩ ⟨порядок⟩
⟨порядок⟩ ::= е { ⟨знак⟩ } ⟨целое⟩
⟨знак⟩ ::= + | —

П о э л е м е н т н о е п р е д с т а в л е н и е используется для обозначения строк, наборов и для обозначения чисел во внутреннем представлении.

⟨поэлементное представление⟩ ::=
1 " ⟨двоичная цифра⟩ ... " | 3 " ⟨восьмеричная цифра⟩ .. "
| 4 " ⟨шестнадцатеричная цифра⟩ ... " | 8 " ⟨литера⟩ ... "

⟨Поэлементное представление⟩ задает смежную последовательность двоичных, восьмеричных, шестнадцатеричных и байтовых элементов. Цифра, предшествующая первой кавычке, определяет размер элемента в разрядах и способ кодировки. Длина последовательности (в разрядах) равна произведению размера элемента на количество элементов.

П о д ч е р к н у т ы е и д е н т и ф и к а т о р ы используются для представления служебных слов. Подчеркнутый идентификатор — это последовательность, состоящая из символа подчеркивания «_», за которым следует ⟨идентификатор⟩. В описании языка подчеркнутые идентификаторы выделены полужирным шрифтом

⟨формат⟩ ::= ⟨простой формат⟩ | ⟨строчный формат⟩
⟨простой формат⟩ : : =**ф32** | **ф64** | **ф128**
⟨строчный формат⟩ : : =**ф1** | **ф4** | **ф8**

Р а з д е л и т е л и — это специальные символы и следующие слитные сочетания:

:= <:= <=> ** :=: +: —: /: *: ^ :
>= <= <> >/ </ >=/ <= / =/ <>/
// && *

2.3. Структура текста программы. Чтобы транслятор мог обработать текст программы, последний представляют в виде файла на каком-либо внешнем носителе, например на перфокартах. Файл текста программы имеет такую же структуру, как и любой другой текстовый файл (подробно см. § 12 гл. 2). Содержимое файла текста про-

граммы строится из строк текста *). Строка представляет собой последовательность базовых символов.

2.4. Комментарии и прагматы. Комментарий может быть коротким и длинным. Короткий комментарий начинается символом % и заканчивается концом той строки, на которой он начался. Непосредственно за символом % не может следовать символ *. Длинный комментарий — это последовательность символов, которая начинается и заканчивается ограничителем длинного комментария. Таким ограничителем является пара символов %*. Длинный комментарий может занимать несколько строк.

Прагматом называется последовательность символов, начинающаяся символом ▢ и заканчивающаяся концом той строки, на которой она началась.

Комментарии полностью игнорируются транслятором и не влияют на выполнение программы. В отличие от этого прагматы могут рассматриваться транслятором, редактором и другими компонентами систем программирования как руководство по режиму трансляции, редактирования и проч.

2.5. Использование пробелов. Пробел может встречаться везде в тексте программы, кроме как внутри лексических элементов. Исключение составляет использование пробела в качестве ⟨литеры⟩ в ⟨поэлементном представлении⟩. Переход с одной строки на другую эквивалентен пробелу, кроме тех случаев, когда он заканчивает короткий комментарий или прагмат. Таким образом, лексический элемент должен располагаться на одной строке. Комментарий может встречаться везде, где разрешен пробел.

Последовательность из двух лексических элементов, первый из которых заканчивается ⟨буквой⟩ или ⟨цифрой⟩, а второй начинается с ⟨буквы⟩ или ⟨цифры⟩, должна быть разделена по крайней мере одним пробелом. В частности, пробелом должны быть разделены подчеркнутый идентификатор и следующие за ним ⟨идентификатор⟩ или ⟨поэлементное представление⟩.

3. Генераторы типов и стандартные типы

Генератор типа создает новый тип массива, области, структуры или объединенный тип. Вновь созданный тип отличен от всех других типов и совпадает только сам с собой. Эта особенность, называемая «эквивалентностью типов по наименованию», существенна при определении семантики контроля типов. Отличие процедурных типов состоит в том, что для этой группы совпадение типов определяется через структурную эквивалентность, т. е. два процедурных типа считаются совпадающими, если совпадают типы параметров и результата.

*) Далее вместо термина «строка текста программы» будем говорить просто «строка».

В этом разделе также описываются средства создания подтипов и стандартные типы, включаемые в стандартный контекст программ пользователей.

3.1. Первичные типы. Первичными типами являются типы массива, области, структуры, модуля и процедуры.

Тип массива. Объект типа массив состоит из элементов, являющихся переменными или константами одного и того же типа.

⟨тип массива⟩ ::=
  | **массив** [{,}. . .] ⟨описание элемента⟩
⟨описание элемента⟩ ::=
  конст {#⟨вычисление типа⟩}
  | **перем** {#⟨вычисление типа⟩}
  | **конст** : ⟨вычисление типа⟩ | ⟨формат⟩

Выполнение ⟨типа массива⟩ заключается в том, что создается новый параметризованный тип массива. Размерность массива данного типа равна числу индексных позиций между скобками. Длины измерений задаются при генерации объекта данного типа или при создании связанного подтипа.

⟨Описание элемента⟩ отличается от похожих по виду предварительного ⟨описания константы⟩ и ⟨описания переменной⟩ тем, что опущен идентификатор. Если в качестве ⟨описания элемента⟩ используется ⟨формат⟩, то элементами массива являются переменные динамического типа. ⟨Описание элемента⟩ вида **перем** и **конст** означают, что элементами массива являются соответственно переменные и константы динамического типа. Инициализация элементов массива, если это необходимо, указывается при его создании с помощью конструкции ⟨генератор объекта⟩ с ⟨доопределением составного⟩.

Тип области. Область состоит из элементов одного и того же типа. Элементы области создаются не в момент создания области, а при создании объекта, тип которого ограничен этой областью.

⟨тип области⟩ ::= **область** [⟨тип длины⟩] ⟨вычисление типа⟩
⟨тип длины⟩ ::= ⟨имя типа⟩ | ⟨отрезок⟩

Выполнение ⟨типа области⟩ заключается в том, что выполняются входящие в его состав ⟨тип длины⟩ и ⟨вычисление типа⟩ и создается новый параметризованный тип области. ⟨Тип длины⟩ используется для того, чтобы указать максимально допустимое количество элементов области. Тип элемента может быть составным, в том числе — параметризованным типом, или типом, объединенным из составных и/или объединенных типов.

Тип структуры. Структура состоит из элементов, являющихся константами или переменными. Тип элементов необязательно должен быть одинаковым.

⟨тип структуры⟩ ::=
структ
{ (⟨последовательность параметров размера⟩)}
({ ⟨последовательность описания элементов структуры⟩})
⟨параметр размера⟩ ::=
⟨список идентификаций параметров длин⟩
| **тег** ⟨список идентификаций параметров выбора⟩
⟨идентификация параметра длины⟩ ::=
⟨идентификатор⟩#⟨тип длины⟩
⟨идентификация параметра выбора⟩ ::=
⟨идентификатор⟩#⟨тип выбора⟩
⟨тип выбора⟩ ::= ⟨имя типа⟩ | ⟨отрезок⟩
⟨описание элемента структуры⟩ ::=
⟨описание константы⟩ { ⟨спецификация размещения⟩}
| ⟨описание переменной⟩ { ⟨спецификация размещения⟩ }

Выполнение ⟨типа структуры⟩ состоит в следующем. Если указаны формальные параметры, то выполняются ⟨типы длин⟩ и ⟨типы выбора⟩ и создается новый параметризованный тип структуры. Типы формальных параметров равны результатам выполнения ⟨типов длин⟩ и ⟨типов выбора⟩ и должны быть подтипами целого типа. Параметры входят в число элементов структур данного типа как элементы-константы. Если формальные параметры не указаны, создается новый связанный тип структуры, отличный от других типов.

⟨Описание коистанты⟩ и ⟨описание переменной⟩, входящие в состав ⟨типа структуры⟩, не могут содержать инициализацию, т. е. имеют вид предварительного описания.

Тип модуля. Объект типа модуль состоит из интерфейсных элементов и скрытых элементов. В ⟨типе модуля⟩ дается полное или предварительное описание каждого интерфейсного элемента. Доопределение интерфейсных элементов и описание используемых при этом скрытых элементов указываются в ⟨теле модуля⟩. В практике могут встретиться еще два случая:

— природа модуля такова, что можно, не используя ⟨тела модуля⟩, сгруппировать все необходимые описания интерфейсных и скрытых элементов в ⟨типе модуля⟩;

— предварительное описание интерфейсного элемента использует идентификаторы скрытых элементов, т. е. описание скрытого элемента должно предшествовать предварительному описанию интерфейсного элемента.

Чтобы упростить программирование этих случаев, допускается указывать в ⟨типе модуля⟩ предварительные или непосредственные описания скрытых элементов. Доопределение предварительно описанных скрытых элементов, а также описание других скрытых элементов (если таковые необходимы) дается в ⟨теле модуля⟩.

⟨тип модуля⟩ ::=
интерфейс
{⟨контекст⟩ **начало**}
⟨секция контекста⟩
конинт

Контекстная приставка ⟨контекст⟩ содержит предварительные или полные описания скрытых элементов модуля (или части скрытых элементов). Контекстной приставки может не быть. Другая часть ⟨типа модуля⟩, а именно ⟨секция контекста⟩, всегда входит в состав этой конструкции. Она содержит предварительные или полные описания интерфейсных элементов.

Выполнение ⟨типа модуля⟩ состоит в том, что создается новый непараметризованный тип.

З а м е ч а н и е. Как и в случае ⟨типа структуры⟩, выполнение ⟨типа модуля⟩ не приводит к выполнению содержащихся в нем описаний. Эти действия производятся при создании объекта такого типа.

Тип процедуры. В ⟨типе процедуры⟩ задаются типы параметров и в случае процедуры-функции — тип результата. При вызове процедур определяемого типа производится контроль параметров. Отличительные особенности процедур стандартного типа описаны в пп. 3.6 и 11.1.

⟨тип процедуры⟩ ::=
проц({⟨последовательность описаний параметров⟩})
| **функция** ({⟨последовательность описаний параметров⟩})
{#⟨имя типа⟩}
⟨описание параметра⟩ ::=
{**прог**} ⟨описание константы⟩
| {**прог**} ⟨описание переменной⟩

⟨Описание константы⟩ и ⟨описание переменной⟩ не должно содержать инициализацию и не должно являться ⟨описанием переменной⟩ с ⟨простым форматом⟩. ⟨Описание параметра⟩ определяет также возможные способы передачи фактических параметров в ⟨вызове⟩ процедуры данного типа (п. 11.1).

Выполнение типа процедуры состоит в том, что выполняются ⟨вычисления типов⟩ из ⟨описания параметров⟩ и ⟨вычисление типа⟩ тела процедуры-функции. Если ⟨вычисление типа⟩ тела не указано, то этим типом является динамический тип. Затем создается тип процедуры, тип параметров и результата, которой равен результатам вычислений типов.

Процедурные типы считаются (статически) совпадающими, если у них (статически) совпадают типы соответствующих параметров и тип тела. При этом не требуется совпадения идентификаторов параметров.

В ⟨описании параметров⟩ можно не указывать ⟨идентификаторы⟩ параметров. В этом случае нужно опускать ⟨идентификаторы⟩ всех параметров.

3.2. Объединенный тип. В типе объединения задаются типы, составляющие объединенный тип.

```
⟨тип объединения⟩ ::=
    выбтип ⟨тип выбора⟩ из
  ⟨список составляющих типов⟩
    всевыб
⟨составляющий тип⟩ ::= ⟨выражение⟩:⟨вычисление типа⟩
```

Выполнение ⟨типа объединения⟩ состоит в том, что выполняются ⟨тип выбора⟩, ⟨выражения⟩ и ⟨вычисления типов⟩ и создается новый параметризованный тип. Параметр выбора должен быть подмножеством целого типа. В состав нового типа входят типы, являющиеся результатом ⟨вычисления типов⟩. Каждому составляющему типу поставлена в соответствие метка в виде целого, являющегося результатом вычисления соответствующего ⟨выражения⟩ статического класса. Целое должно быть ⟨типа выбора⟩. Составляющие типы не могут пересекаться, т. е. не должны пересекаться соответствующие множества. Все метки должны различаться.

3.3. Вычисление типов. Конструкция ⟨вычисление типа⟩ определяет допустимые в языке виды выражений, значениями которых являются типы.

```
⟨вычисление типа⟩ ::=
    ⟨изображение типа⟩ | ⟨вычисление подтипа⟩
  | ⟨имя типа⟩
⟨изображение типа⟩ ::=
    ⟨тип массива⟩ | ⟨тип области⟩
  | ⟨тип структуры⟩ | ⟨тип модуля⟩
  | ⟨тип процедуры⟩ | ⟨тип объединения⟩
⟨вычисление подтипа⟩ ::=
    ⟨отрезок⟩ | ⟨перечисление⟩ | ⟨элемент области⟩
  | ⟨связывание типа⟩ | ⟨выбор типа⟩
⟨имя типа⟩ ::=
    ⟨идентификатор⟩ | ⟨элемент модуля⟩
    | ⟨внешнее имя⟩
```

Результатом ⟨вычисления типа⟩ является результат выполнения ⟨изображения типа⟩, ⟨вычисления подтипа⟩ или тип, являющийся результатом выполнения ⟨имени типа⟩. Значением ⟨имени типа⟩ может быть:

— тип, являющийся значением константы типа *тип*, обозначенной ⟨идентификатором⟩ или ⟨элементом модуля⟩;

— тип, являющийся программой-объектом. В этом случае ⟨внешнее имя⟩ должно обозначать указатель, установленный на элемент справочника, ссылающийся на данную программу-объект, или на файл объектного кода, содержащий данную программу-объект (см. § 12).

Подтип целого. Подтип целого задается с помощью ⟨отрезка⟩ или ⟨перечисления⟩.

⟨отрезок⟩ ::= до ⟨выражение⟩

Результатом выполнения ⟨отрезка⟩ является тип подмножества целого, включающего числа от 0 до значения ⟨выражения⟩. Это ⟨выражение⟩ задает верхнюю границу отрезка и должно быть статического класса типа целое. Если верхняя граница не превышает максимально допустимого значения типа *цел*, то ⟨отрезок⟩ задает подтип типа *цел*, а иначе — подтип типа *дцел*.

⟨перечисление⟩ ::= (⟨список идентификаторов⟩)

Результатом выполнения ⟨перечисления⟩ является тип подмножества целого, включающего числа от 0 до n—1, где n — число идентификаторов. ⟨Перечисление⟩ задает подтип типа *цел*. ⟨Список идентификаторов⟩ выполняет роль описаний констант вновь созданного типа. Значением констант являются элементы подмножества. Значения констант упорядочены в соответствии с расположением идентификаторов в списке.

⟨Перечисление⟩ можно использовать только в правой части ⟨описания типа⟩. Область действия идентификаторов совпадает с областью действия ⟨описания типа⟩, правой частью которого является данное ⟨перечисление⟩.

Связанные подтипы. Конструкция ⟨связывание типа⟩ задает подтип составного типа, получаемый путем связывания параметров типа.

⟨связывание типа⟩ ::=
⟨имя типа⟩ ((список выражений))
| ⟨имя типа⟩ ((список типов индексов))
| **массив** [⟨список выражений⟩] ⟨описание элемента⟩
| **массив** [⟨список типов индексов⟩] ⟨описание элемента⟩
⟨тип индекса⟩ ::= ⟨имя типа⟩ | ⟨отрезок⟩

Результатом выполнения ⟨имени типа⟩ должен являться составной параметризованный тип, являющийся первичным типом или его подтипом, ограниченным областью. Значения ⟨выражений⟩ задают фактические параметры типа. В результате выполнения ⟨связывания типа⟩ создается новый тип, являющийся связанным подтипом исходного типа.

Для связывания типа массива можно использовать либо ⟨список выражений⟩, который задает длины измерений, либо ⟨список типов индексов⟩, который задает максимально допустимое значение индексов по измерениям (максимальное значение индекса на 1 меньше длины измерения). ⟨Тип индекса⟩ должен быть подтипом типа *цел*.

Выполнение ⟨связывания типа⟩ вида

массив [*е*1, *е*2, . . ., *ек*] ОЭ или **массив** [*т*1, *т*2, . . ., *тк*] ОЭ

где *е*1, . . ., *ек* — ⟨выражения⟩, *т*1, *т*2, . . ., *тк* — подтипы типа *цел*, а ОЭ — ⟨описание элемента⟩, эквивалентно ⟨вычислению типа⟩ вида

тм (*е*1, *е*2, . . ., *ек*) или *тм* (*т*1, *т*2, . . . *тк*)

в контексте ⟨описания типа⟩

тип *тм* = **массив** [, , . . .,]ОЭ

Выбор типа. Конструкция ⟨выбор типа⟩ производит выбор одного из типов, составляющих объединенный тип.

⟨выбор типа⟩ ::=
⟨имя типа⟩ (⟨выражение⟩)
|**выбтип** ⟨выражение⟩ **из**
⟨список составляющих типов⟩
всевыб

Результатом выполнения ⟨имени типа⟩ должен являться объединенный тип. Результатом ⟨выбора типа⟩ является тип, входящий в состав объединенного типа и имеющий метку, равную значению ⟨выражения⟩.

Выполнение ⟨выбора типа⟩ вида

выбтип *е* **из** *м*1 : *т*1 . . ., *мп* : *тп* **всевыб**

эквивалентно ⟨вычислению типа⟩ вида *тв* (*е*) в контексте ⟨описания типа⟩

тип *тв* = **выбтип** *т* **из** *м*1: *т*1, . . ., *мп* : *тп* **всевыб**

где *т* — подтип целого, включающий целые числа *м*1, . . ., *мп*, а *е* — ⟨выражение⟩ статического класса типа *т* или ⟨идентификатор⟩ формального параметра выбора типа *т*.

Тип ограниченный областью. Конструкция ⟨ограничение областью⟩ производит вычисление подтипа, ограниченного областью:

⟨ограничение областью⟩ ::= **элем** ⟨идентификатор⟩

⟨Идентификатор⟩ должен обозначать константу типа области. Если *т* является типом элемента области, то в результате выполнения ⟨ограничения областью⟩ создается новый тип, являющийся подтипом

типа m, ограниченным заданной областью. Объекты, имеющие такой тип, принадлежат области. ⟨Ограничение областью⟩ можно использовать только в правой части ⟨описания типа⟩.

3.4. Фактические параметры типов, соотношение типов в присваивании. Для того чтобы избежать скрытого интерпретируемого контроля типов *), наложен ряд ограничений на класс ⟨выражений⟩, задающих фактические параметры типов, а также на тип правых частей присваивания и схожих с ним конструкций:

1. Если формальный параметр является параметром выбора, то ⟨выражение⟩, задающее значение этого параметра, должно быть либо выражением статического класса, либо ⟨идентификатором⟩ формального параметра выбора, описанным в непосредственно охватывающем ⟨типе структуры⟩.

2. Если формальный параметр является параметром длины, то значение этого параметра можно задавать с помощью ⟨выражения⟩ динамического класса. В этом случае результат ⟨вычисления типа⟩ называется динамически связанным типом. ⟨Выражение⟩ динамического класса можно использовать только в правой части ⟨описания типа⟩ и в ⟨генераторе объекта⟩. В остальных случаях ⟨выражение⟩ должно быть статического класса или являться ⟨идентификатором⟩ формального параметра длины, описанным в непосредственно охватывающем ⟨типе структуры⟩.

3. ⟨Идентификатор⟩ формального параметра типа можно использовать только для описания типов встроенных констант.

4. На типы встроенных констант накладываются ограничения, зависящие от типа объекта, элементом которого является константа:

— тип массива или совокупность формальных параметров процедуры. Тип константы должен быть статически связанным типом структуры или вектора;

— тип структуры. Тип константы должен быть статически связанным типом структуры или вектора или типом, зависящим от формального параметра типа непосредственно объемлющей структуры, причем после подстановки параметра тип константы должен становиться связанным типом структуры или вектора;

— тип модуля или совокупность локальных описаний процедуры или тела модуля. Тип константы должен быть статически связанным типом структуры или массива, типом модуля или типом процедуры.

Если требуется, чтобы тип константы был статически связанным типом, то встроенные константы, являющиеся элементами значения

*) Под этим термином понимаются такие виды динамического контроля типов, которые не отражены явным образом в тексте программы, но которые необходимо в процессе выполнения программы производить интерпретационно, т. е. без специальной аппаратной поддержки.

этой константы, также должны удовлетворять этому ограничению. Тип встроенной константы не может быть типом, ограниченным областью.

5. В определениях процедурных типов, соотношения типов в присваивании и эквивалентных типов используются приводимые ниже понятия совпадения типов и принадлежности одного типа другому типу.

Совпадение типов. Типы *та* и *тв* (статически) совпадают, если выполняются следующие условия:

та и *тв* — (статически) один и тот же тип, или

та и *тв* — (статически) связанные подтипы типов *тах* и *твх*, и *тах* и *твх* (статически) совпадают, и соответствующие параметры *та* и *тв* попарно равны, или

та и *тв* подтипы типов *тах* и *твх*, ограниченные областями *а* и *в* соответственно, и *тах* и *твх* (статически) совпадают, и области *а* и *в* — (статически) одна и та же область.

Замечание. Если две разные модульные константы одинакового типа имеют одноименные элементы типа *тип*, то считается, что значениями этих элементов являются статически разные типы. Это правило, связанное со статическим контролем типов, не учитывает, что фактически значениями модульных констант может быть один и тот же объект, и в этом случае указанные элементы суть один и тот же элемент и, следовательно, один и тот же тип. Аналогично, при статическом контроле типов считается, что значением двух констант типа области являются разные объекты. При необходимости фактическое совпадение типов может быть динамически проверено с помощью конструкции ⟨квалификация⟩.

Принадлежность типов. Тип *та* (статически) принадлежит типу *тв*, если выполняются следующие условия:

та и *тв* (статически) совпадают, или

та является подтипом *тв*, или

та и *тв* — (статически) связанные подтипы типов *тах* и *твх*, и *тах* (статически) принадлежит *твх*, и соответствующие параметры типов попарно равны, или

та и *тв* — подтипы типов *тах* и *твх*, ограниченные областями *а* и *в*, и *тах* (статически) принадлежит *твх* и области *а* и *в* — (статически) одна и та же область.

6. Тип ⟨выражения⟩, значение которого присваивается переменной (становится значением константы, становится значением параметра длины или выбора), должен статически принадлежать типу переменной (типу константы, типу параметра длины или параметра выбора).

Отступление от этого ограничения допускается для типа целых. В позиции, где требуется некоторый тип *т*, являющийся подтипом типа *цел* (*дцел*), можно использовать выражение *е*, тип которого яв-

ляется другим подтипом типа *цел (дцел)* или типом *цел (дцел)*. В этом случае проверяется, возможно, динамически, принадлежит ли значение ⟨выражения⟩ требуемому подмножеству целого, т. е. выполнение ⟨выражения⟩ *e* эквивалентно выполнению ⟨квалификации⟩ *e#m*.

7. В ряде конструкций допускается преобразование типа объекта к эквивалентному типу. Тип *та* массива или структуры эквивалентен типу *тв*, если типы соответствующих параметров и элементов (в порядке перечисления в генераторе типа), являющихся отдельно размещенными константами или переменными, попарно статически совпадают, а типы элементов, являющихся встроенными константами, попарно эквивалентны.

3.5. Описание типа. Описание типа вводит константу типа *тип*.

⟨описание типа⟩ ::= **тип** ⟨список идентификаций типов⟩
⟨идентификация типа⟩ ::= ⟨идентификатор⟩ {= ⟨вычисление типа⟩}

При выполнении описания с правой частью, имеющей вид ⟨вычисления типа⟩, создается константа, значением которой является результат ⟨вычисления типа⟩.

Описание типа без правой части является предварительным описанием типа. Для типа, введенного с помощью предварительного описания, должно быть дано доопределение типа, которое имеет вид ⟨описания типа⟩ с тем же идентификатором и с правой частью. Накладывается ряд ограничений на использование ⟨идентификатора⟩ типа в описаниях, предшествующих доопределению этого идентификатора:

— ⟨идентификатор⟩ можно использовать только для указания типов переменных и отдельно размещенных констант;

— ⟨идентификатору⟩ нельзя придавать фактические параметры типа (в конструкциях ⟨связывание типа⟩ и ⟨выбор типа⟩);

— если ⟨идентификатор⟩ использован в правой части описания другого типа, то последний тип называется незавершенным. Незавершенные типы также можно использовать только для указания типа переменных и отдельно размещенных констант.

Предварительное описание типа используется в двух случаях: для описания скрытого типа и для описания взаимозависимых типов.

Скрытые типы. Предварительное описание скрытого типа входит в состав описаний ⟨типа модуля⟩. Доопределение типа должно быть дано в ⟨теле модуля⟩. Вне ⟨тела модуля⟩ запрещены следующие действия со скрытым типом; нельзя использовать элементы объекта скрытого типа, т. е. их нельзя обозначать посредством конструкций ⟨элемент массива⟩, ⟨элемент структуры⟩ и ⟨элемент модуля⟩; скрытый тип нельзя использовать в ⟨генераторе объекта⟩, в описании встроенной константы и в операции **тег**.

3.6. Стандартные типы объектов. Стандартные типы описаны в интерфейсе модуля стандартного контекста. Программы пользова-

телей транслируются и выполняются в контексте этого модуля.

Наборы. Базовым типом, из которого конструируются другие типы, является тип *лог*, описанный как скрытый тип. Наборы представляют собой последовательности (массивы) объектов типа *лог*. Длина этой последовательности называется длиной набора. Длина набора не превышает 64 элементов.

```
тип лог,
    наб = выбтип до 64 из
        0 : массив [0] конст#лог,  % пустой набор
        1 : лог,
        2 : массив [2] конст#лог,  % набор длины 2
        % и т. д. Альтернативы имеют вид:
        % i : массив [i] конст#лог  % набор длины i
        64 : массив [64] конст#лог  % полный набор
        всевыб,
    циф = наб (4),  % тип «цифра»
    лит = наб (8);  % тип «литера».
```

Набор имеет несколько интерпретаций:

а) В ряде предложений набор обрабатывается как последовательность элементов того или иного строчного формата:

1. Логическое, т. е. **истина** или **ложь**.

2. Битовое, т. е. последовательность элементов типа *лог* формата ф1.

3. Цифра, т. е. один элемент формата ф4, полученный путем сцепления четырех последовательных элементов типа *лог*.

4. Литера, т. е. один элемент формата ф8, полученный путем сцепления восьми последовательных элементов типа *лог*.

5. Цифровое, т. е. последовательность элементов формата ф4, заполненных шестнадцатеричными цифрами.

6. Литерное, т. е. последовательность элементов формата ф8, заполненных целыми, величина которых заключается в диапазоне от 0 до 255, в том числе — кодами литер.

б) Предложения, определенные для целочисленных операндов, интерпретируют набор как целое без знака (при этом считается, что набор содержит двоичное представление целого). Тем самым обеспечивается автоматическое преобразование из битовых, цифровых и литерных в целое.

Изображение набора. Набор задается с помощью ⟨поэлементного представления⟩. Длина набора равна длине ⟨поэлементного представления⟩ и не должна превышать 64 разрядов.

⟨набор⟩ ::= ⟨поэлементное представление⟩ . . . | **истина** | **ложь** |""'

Результирующий набор — это объект статически связанного типа *наб* (*к*), где *к* — длина набора.

Значения типа *лог* (или *наб* (1)) обозначаются как **истина** и **ложь**. Две слитные кавычки'''' обозначают пустой набор, т. е. набор нулевой длины (*наб* (0)). При изображении литерного набора разрешается опускать цифру 8, предшествующую первой открывающей кавычке.

■ Допускается смешанное представление набора с помощью нескольких поэлементных представлений с различным форматом элементов. В этом случае изображение последовательности независимо от формата элементов плотно сцепляются и образуют набор. ■

Целое. Объекты целых типов представляют собой целые числа со знаком. Целые числа могут быть двух форматов — ф32 (тип *цел*), ф64 (тип *дцел*).

```
    % целое формата ф32:
тип
цел = структ (конст зн#лог, мант : наб(31)),
    % целое формата ф64:
дцел = структ (конст зн#лог, мант : наб(63)),
    % универсальный класс целых формата ф32 и ф64:
тунц = (тунцц, тунцдц, тунцн),
унцел =
выбтип тунц из тунцц : цел, тунцдц : дцел, тунцн : наб всевыб;
    % максимальные целые числа:
конст максцел#цел, максдцел#дцел;
```

Вещественное. Вещественные числа могут быть трех форматов — ф32 (тип *квещ*), ф64 (тип *вещ*), ф128 (тип *двещ*).

```
    % длины мантисс вещественных чисел:
конст длмквещ#цел=24, длмвещ#цел=56;
    % короткое вещественное формата ф32:
тип
квещ = структ (конст зн#лог, порядок : цел,
                     мант : наб (длмквещ)),
    % вещественное формата ф64:
вещ = структ (конст зн#лог, порядок : цел,
                     мант : наб (длмвещ)),
    % длинное вещественное формата ф128:
двещ = структ (конст зн#лог, порядок : цел,
                     мантмл :,мантст : ,наб (длмвещ));
    % максимальные вещественные числа и
    % вещественные нули:
конст максвещ#, миндвещ#двещ;
```

Изображение числа.

⟨число⟩ :: =
    { ⟨тег целого⟩ } ⟨целое⟩
  | { ⟨тег вещественного⟩ } ⟨вещественное⟩
⟨тег целого⟩ ::= **цел32** | **цел64**
⟨тег вещественного⟩ ::= **вещ32** | **вещ64** | **вещ128**

Тип (и формат) числа определяется тегом, указанным в начале изображения: **цел32** — *цел*, **цел64** — *дцел*, **вещ32**—*квещ*, **вещ64** — *вещ*, **вещ128** — *двещ*. Если тег опущен, то в зависимости от контекста для целых выбирается тип *цел* или *дцел*, а для вещественных — тип *квещ*, *вещ* или *двещ* *).

Числовые типы. Арифметические операции определены не только для целых и вещественных типов, но и для числовых типов, которые представляют собой объединение из целых и вещественных типов. Тип *кчис* объединяет целые и вещественные формата ф32, тип *чис* объединяет целые и вещественные формата ф64, тип *унчис* объединяет все стандартные целые, вещественные и наборы.

```
    % короткое числовое формата ф32:
тип
ткч = (ткчц, ткчкв),
кчис = выбтип ткч из ткчц : цел, ткчкв : квещ всевыб,
    % числовое формата ф64:
тч = (тчдц, тчв),
чис = выбтип тч из тчдц : дцел, тчв : вещ всевыб,
    % универсальный числовой тип всех форматов:
тунч - (тунчкч, тунчч, тунчн, тунчдв),
унчис = выбтип тунч из
                    тунчкч : кчис, тунчч : чис,
                    тунчн : наб, тунчдв : двещ
        всевыб;
```

Пустой объект. Этот объект в основном используется для представления начального значения неинициализированных переменных. Кроме того, его можно присваивать и передавать как параметр; он допускается в операциях проверки типа. В остальных случаях использование этого объекта приводит к ошибке.

■ Ошибка, связанная с использованием пустого объекта, приводит к возникновению ситуации *неверныйоперанд*. Ошибка возникает при выполнении операций над таким объектом, а не в момент выборки его из переменной. ■

Изображение пустого объекта используется для обозначения неинициализированных данных:

*) С помощью прагматов транслятор может допускать возможность выбора другого умолчания.

⟨пустой объект⟩ ::= **пусто32** | **пусто64** | **пусто** ⟨вычисление типа⟩

где **пусто32** обозначает пустой объект формата ф32, **пусто64** — пустой объект формата ф64, **пусто** ⟨вычисление типа⟩ — пустой объект заданного первичного типа.

Массив. Стандартные типы массивов разбиты на две группы: смежные векторы (с-векторы) и выстроенные массивы (в-массивы).

С м е ж н ы й в е к т о р. Элементы такого вектора размещаются в памяти смежно. Над с-векторами определен ряд встроенных в язык групповых операций, таких как: пересылка, поиск элемента, имеющего заданное значение, сравнение.

```
тип тэм = (тлог, тциф, тлит, ткор, тдин, тдлин),
    % типы с-векторов констант
    стстр = выбтип тэм из
            тлог : массив [ ] конст#лог,
            тциф : массив [ ] конст#циф,
            тлит : массив [ ] конст#лит,
            ткор : массив [ ] конст#кчис,
            тдин : массив [ ] конст,
            тдлин : массив [ ] конст#унчцо
            всевыб,
    % типы с-векторов переменных
    стсв = выбтип тэм из
            тлог : массив [ ] ф1,
            тциф : массив [ ] ф4,
            тлит : массив [ ] ф8,
            ткор : массив [ ] ф32,
            тдин : массив [ ] ф64,
            тдлин : массив [ ] ф128
            всевыб,
    тствект = (тстр, тсв),
    ствект = выбтип тствект из
            тстр : стстр,
            тсв : стсв
            всевыб;
```

В дальнейшем, если особо не оговорено, под термином «вектор» понимаются массивы типа *ствект* и одномерные массивы программно определяемых типов. Одномерные выстроенные массивы описанных ниже стандартных типов в эту группу не включены.

Выстроенный массив может содержать n измерений ($n \geqslant 1$). Элементы в-массива размещены в памяти так, что смещение элемента относительно начала массива линейно зависит от его индексов. Отсюда название «выстроенный массив».

```
тип
% типы в-массивов
вм1 = выбтип тэм из
        тлог : массив [ ] ф1,
        тциф : массив [ ] ф4,
        тлит : массив [ ] ф8,
        ткор : массив [ ] ф32,
        тдин : массив [ ] ф64,
        тдлин : массив [ ] ф128
        всевыб,
% аналогично строятся описания типов двухмерного массива вм2
% и т. д. до вм7.
ствм = выбтип до 7 из
1 : вм1, 2 : вм2, 3 : вм3, 4 : вм4,5 : вм5, 6 : вм6, 7 : вм7
    всевыб;
```

Отличие одномерного в-массива (в-вектора) от вектора проявляется в следующем. С точки зрения использования вектор представляет больше возможностей, чем выстроенный вектор. Например, существует ряд групповых операций, определенных только для векторов: пересылка, поиск элемента, сравнение. С другой стороны, в-вектор реализует общий случай представления в памяти: элементы такого вектора могут располагаться несмежно на одинаковом расстоянии друг от друга. Такую структуру имеет одномерное сечение многомерного массива.

Изображение вектора. Конструкция ⟨статический вектор⟩ предназначена для непосредственного изображения с-векторов стандартных типов в тексте программы. Значение элементов статического вектора не изменяется в процессе выполнения программы.

■ Попытка изменить значение приводит к возникновению ситуации *нарушениезащиты.* ■

⟨статический вектор⟩ ::= ⟨массив констант⟩ | ⟨строка⟩

⟨Строка⟩ — это изображение логического, цифрового или литерного статического вектора статически связанного типа, состоящего из *к* элементов, заданных между кавычками. Элементы нумеруются слева направо, начиная с нуля. Тип массива определяется в зависимости от служебного слова *): **стр1** — *стрлог*(*к*), **стр4** — *стрциф*(*к*), **стр8** — *стрлит*(*к*).

⟨строка⟩ ::=
    стр1 ⟨поэлементное представление⟩. . .
  | **стр4** ⟨поэлементное представление⟩. . .
  | **стр8** ⟨поэлементное представление⟩. . .

*) Здесь и далее используются сокращенные обозначения типов с-векторов (см. Приложение 6).

■ ⟨Поэлементные представления⟩ сцепляются таким же образом, как и в случае наборов. Допускается смешанное представление строки. Кодировку формата в первом ⟨поэлементном представлении⟩ можно опустить в случае, если она совпадает с результирующим форматом. Например, строка **стр8** 8''*литеры*'' эквивалентна строке **стр8**''*литеры*''. Путем разбиения ⟨строки⟩ на ⟨поэлементные представления⟩ ее можно переносить с одной строки текста программы на другую. ■

Процедуры стандартного типа. В стандартном контексте описан один скрытый стандартный тип *стпроц*, который является стандартным типом процедуры. Процедуры этого типа обладают следующими особенностями:

— при их вызове не производится контроль параметров;

— они могут иметь переменное число параметров.

Способ изображения таких процедур описан в п. 11.1.

■ *Метка перехода.* Этот объект скрытого типа *стмпход* используется при передаче управления как указатель помеченного предложения программы. ■

Ситуация. Ситуация — это средство идентификации обстоятельств, при которых происходит прекращение выполнения закрытых предложений (см. § 1). С целью оптимизации различаются статические и динамические ситуации. В первом случае ситуация и соответствующая реакция задаются статически. Во втором случае ситуация создается генератором, и реакцию можно динамически переопределять.

■ В процессе выполнения операций, встроенных в язык, могут быть обнаружены ошибки, например, переполнение результата арифметической операции, или попытка чтения за границей файла. Для подобных случаев предусмотрены стандартные ситуации. В интерфейсе модуля стандартного контекста описаны константы, значениями которых являются эти ситуации. В описании семантики операций указываются обстоятельства, при которых возникают ситуации, и приводятся идентификаторы соответствующих констант. Поскольку идентификаторы известны, программа может задать собственную реакцию на ошибочную ситуацию (подробнее см. § 10).

Ниже, в описании семантики конструкций, допускающих операнды динамического типа, приводятся ограничения на возможные типы значений этих операндов. Если при выполнении программы эти ограничения нарушаются и иное не оговорено, то подразумевается, что возникает ситуация *неверныйоперанд*.

Статическая ситуация — это константа, значением которой является метка перехода. Динамическая ситуация — это объект скрытого типа *стсит*. ■

Внешние объекты. Внешним объектом является файл, контейнер или справочник. Файл содержит данные или программу. Контейнер

содержит совокупность файлов. Справочник предназначен для организации архива. Взаимодействие программы с внешним объектом осуществляется посредством оперативных объектов связи. Поэтому будем говорить, что оперативный объект связи приводит к внешнему объекту.

Оперативные объекты связи. Существуют следующие скрытые стандартные типы оперативных объектов связи:

— указатель внешнего объекта (*стув*), используемый при поиске по архивным справочникам;

— заголовок открытого файла (*стфайл*), заголовок открытого контейнера (*стконт*) и блок ввода/вывода (*стбвв*), предназначенные для работы с объектами в открытом состоянии;

— позиционная переменная (*стпозп*) и таблица буферов (*стбуф*), используемые в процессе буферизованного обмена с файлом.

Все перечисленные объекты создаются динамически.

■ *Семафор.* Объект скрытого типа семафор (*стсем*) используется при синхронизации параллельных процессов.

Паспорт в-массива. Доступ к в-массиву осуществляется посредством промежуточного объекта скрытого типа «паспорт выстроенного массива» (*стпаспвм*). Создание массива сопровождается неявным созданием паспорта. При программировании знание о существовании такого объекта нужно лишь в случае обработки массивов со статически неизвестным числом измерений, а также при использовании конструкции ⟨формирование паспорта⟩ для изменения формы массива. ■

4. Генератор объекта

Конструкция ⟨генератор⟩ предназначена для динамического создания объектов. В общем случае объект любого программно определяемого или стандартного типа, не являющегося скрытым типом, может быть создан с помощью ⟨генератора объекта⟩, в котором указан тип создаваемого объекта. Массив стандартного типа может быть создан как с помощью ⟨генератора объекта⟩, так и с помощью специального ⟨генератора массива⟩. Остальные виды ⟨генераторов⟩ предназначены для создания объектов скрытых стандартных типов.

⟨генератор⟩ ::=
⟨генератор объекта⟩ | ⟨генератор массива⟩
| ⟨генератор ситуации⟩ | ⟨генератор объекта связи⟩
| ⟨генератор внешнего объекта⟩ | ⟨генератор паспорта⟩
⟨генератор объекта⟩ ::=
ген ⟨вычисление типа⟩ {[⟨список атрибутов⟩]}
{ **иниц** ⟨доопределение объекта⟩}
⟨доопределение объекта⟩ ::=

⟨доопределение составного⟩ | ⟨доопределение модуля⟩
| ⟨доопределение процедуры⟩
⟨атрибут⟩ ::= *локал* : ⟨выражение⟩ | *вгвект* : ⟨выражение⟩

Конструкция выполняется следующим образом. Выполняется ⟨вычисление типа⟩. Результатом должен быть связанный составной или процедурный тип. Затем создается объект данного типа и выполняется ⟨доопределение объекта⟩, в результате чего доопределяются элементы составного объекта или тело процедуры. Если тип ограничен некоторой областью, то вновь созданный объект становится элементом этой области.

Создание объекта выполняется в зависимости от типа:

— тип структуры. Создаются элементы-константы, являющиеся параметрами длины и выбора. Значениями этих констант становятся соответствующие значения фактических параметров типа. Затем выполняются заданные в типе структуры предварительные описания остальных элементов, являющихся константами, и описания переменных;

— тип массива. Создается n-мерный массив, состоящий из $к1 * к2 * \ldots * кn$ элементов, где $к1, \ldots, кn$ — значения параметров длины. Каждый из элементов создается путем выполнения заданного в ⟨типе массива⟩ предварительного описания константы или описания переменной. Индекс по p-му измерению может заключаться в пределах от 0 до $к_p - 1$;

— тип области. Создается пустая область. Элементы области можно создавать путем генерации объектов, тип которых ограничен данной областью;

— тип модуля. Элементы модуля создаются путем выполнения ⟨описаний и включений⟩, заданных в ⟨типе модуля⟩. Областью действия этих описаний является ⟨тип модуля⟩, а также соответствующее ⟨тело модуля⟩;

— тип процедуры. Создается процедурный объект с пустым телом.

Время жизни объекта. Если атрибут *локал* не задан, то объект является локальным. В противном случае время жизни объекта задается значением атрибута *локал*.

Для управления временем жизни используются объекты стандартного скрытого типа *ствж*. Объект такого типа создается процедурой *генвж* и всегда является локальным. Если объект x типа *ствж* присваивается атрибуту *локал*, то вновь создаваемый объект имеет такое же время жизни, как и объект x. Если этому атрибуту присваиваются значение ложь, то объект становится глобальным.

В других генераторах атрибут *локал* используется аналогичным образом. Если этот атрибут не указан в ⟨генераторе⟩, то создается локальный объект.

Числа и наборы, в том числе создаваемые с помощью ⟨генератора объекта⟩, имеют неограниченное время жизни.

■ Попытка доступа к уничтоженному объекту приводит к возникновению ситуации *ситметмас*.

Уничтожение объекта можно осуществить явно с помощью стандартной процедуры *откреп*, имеющей один параметр динамического типа:

откреп (*а*)

где: *а* — уничтожаемый объект.

Вектор плавающей длины. В генераторе вектора может быть задан атрибут *вгвект*. В этом случае создается вектор плавающей длины. Значение параметра типа вектора задает начальное количество элементов, а значение атрибута *вгвект* — максимально возможное число элементов. Изменение длины вектора осуществляется с помощью стандартной функции *измдлину*. Эта функция имеет два параметра

измдлину (*а*, *к*)

где: значение параметра *а* — вектор плавающей длины, а значение *к* — целое положительное число, задающее приращение количества элементов вектора. Значением функции является результирующий вектор с увеличенным количеством элементов.

Первый параметр можно передавать не только значением, но также именем

измдлину (имя *а*, *к*)

где: *а* обозначает переменную (или отдельно размещенную константу), значением которой является вектор. После выполнения функции новым значением переменной (или константы) становится результирующий вектор. Это позволяет использовать вызов этой функции в качестве ⟨оператора⟩. ■

4.1. Доопределение структур и массивов. В ⟨доопределении составного⟩ задается значение элементов структуры или массива.

⟨доопределение составного⟩ ::=
⟨выражение⟩ | (⟨список доопределений элементов⟩)
⟨доопределение элементов⟩ ::=
{{⟨обозначение⟩: , }... ⟨обозначение⟩} : ⟨выражение⟩
| {{⟨обозначение⟩: , }... ⟨обозначение⟩} : ⟨доопределение объекта⟩
| [⟨диапазон⟩] : ⟨доопределение объекта⟩
⟨обозначение⟩ ::= ⟨идентификатор⟩ | ⟨выражение⟩

Первая альтернатива правила для ⟨доопределения составного⟩ задает доопределение объекта путем копирования другого объекта,

являющегося значением ⟨выражения⟩. Ниже доопределяемый объект обозначается x, а копируемый — y. Типы x и y должны быть статически связанными, эквивалентными типами. Процесс копирования определяется рекурсивным правилом: если элемент x — переменная или отдельно размещенная константа, его значением становится значение соответствующего элемента y, а если элемент x — встроенная константа, то объект, являющийся значением элемента, доопределяется путем копирования объекта, являющегося значением соответствующего элемента y.

Вторая альтернатива правила для ⟨доопределения составного⟩ задает доопределение посредством формирования. Такой вид доопределения задает значения каждого элемента доопределяемого объекта по отдельности.

Вид ⟨обозначения⟩ элемента зависит от типа доопределяемого объекта: ⟨идентификатор⟩ служит для обозначения элемента структуры, ⟨выражение⟩—для обозначения элемента одномерного массива. Список ⟨обозначений⟩ применяется для того, чтобы задать одно и то же значение для всех элементов, перечисленных в этом списке. ⟨Доопределение элемента⟩ с опущенной левой частью обозначает все элементы доопределяемого массива, не вошедшие в число обозначенных с помощью индексов и ⟨диапазонов⟩. Такое доопределение может быть только последним элементом списка доопределений.

⟨Диапазон⟩ обозначает подмассив доопределяемого одномерного массива (см. п. 6.4.). Доопределение этого подмассива происходит по правилам доопределения встроенной константы типа массива, т. е. либо путем копирования другого объекта, либо посредством формирования. Формирование должно иметь вид: (: ⟨доопределение объекта⟩), т. е. всем элементам подмассива придается одно и то же значение.

Правая часть ⟨доопределения элемента⟩ выбирается в зависимости от разновидности элемента:

— отдельно размещенная константа или переменная (первая альтернатива правила). Результат выполнения ⟨выражения⟩ становится значением элемента;

— встроенная константа (вторая альтернатива правила). Выполняется доопределение объекта, являющегося значением элемента.

4.2. Генератор массива. Массив стандартного типа может быть создан не только универсальным ⟨генератором объекта⟩, но также с помощью ⟨генератора массива⟩.

⟨генератор массива⟩ ::=
  ⟨генератор в-массива⟩ | ⟨генератор с-вектора⟩
⟨генератор в-массива⟩ ::=
  ⟨локализация⟩ [⟨список выражений⟩] ⟨формат⟩
⟨генератор с-вектора⟩ ::=

⟨локализация⟩ **вект** [⟨выражение⟩{: ⟨выражение⟩}] ⟨формат⟩
⟨локализация⟩ ::= **лок** | **глоб**

В результате выполнения ⟨генератора в-массива⟩ создается массив типа *ствм* (*n*) (*тэл*) ($к_1$, . . ., $к_n$), где *n* — число ⟨выражений⟩ в скобках, *тэл* выбирается в зависимости от ⟨формата⟩, а $к_1$, . . ., $к_n$ — значения этих ⟨выражений⟩. В результате выполнения ⟨генератора с-вектора⟩ создается вектор типа *стсв* (*тэл*), где *тэл* выбирается в зависимости от ⟨формата⟩.

Локализация. Компонента ⟨локализация⟩ определяет время жизни созданного массива: лок означает, что массив является локальным, а глоб — глобальным.

■ *Вектор плавающей длины.* При выполнении ⟨генератора с-вектора⟩ с двумя ⟨выражениями⟩ создается вектор плавающей длины. Значение первого ⟨выражения⟩ задает начальное количество элементов, а значение второго ⟨выражения⟩ — максимально возможное число элементов. Изменение длины вектора осуществляется с помощью стандартной функции *измдлину*. ■

5. Описания

⟨Описание⟩ вводит переменную или константу и связывает с ней идентификатор. Диапазон текста программы, внутри которого эта связь существует, называется областью действия описания. В области действия описания переменная или константа обозначается идентификатором, связанным с ней в результате выполнения описания.

В программе может быть несколько описаний с одним и тем же идентификатором. За счет текстуальной вложенности конструкций области действия подобных описаний могут перекрываться. Тогда идентификатор в данной точке программы понимается в том смысле, который ему придан в минимальной охватывающей эту точку области действия.

Конструкция ⟨включение в контекст⟩ позволяет распространить область действия описаний интерфейсных элементов модуля на заданный участок текста программы. Особенность, отличающая эту конструкцию от обычного ⟨описания⟩, заключается в том, что она не создает новых переменных или констант, но расширяет область действия описаний переменных и констант, ранее созданных в качестве интерфейсных элементов модуля.

Контекстом идентификаторов для данной точки текста программы называется множество идентификаторов, описание которых действует в этой точке.

С помощью ⟨описаний⟩ вводятся простые переменные (п. 5.2) и следующие константы: константы типа *тип* (п. 3.5), простые констан-

ты (пп. 5.1; 11.1), статические ситуации (предварительное описание, п. 10.1), метки предложений (§ 19), описатели полей (гл. 5) и текстовые макросы (§ 17).

⟨Описание⟩ может встречаться в составе последовательности описаний в начале ⟨закрытого предложения⟩ или ⟨последовательного предложения⟩ и в составе последовательности описаний, заданных в ⟨типе модуля⟩ и ⟨теле модуля⟩. Кроме того, существуют специфические описания, являющиеся составной частью связанных с ними конструкций. К ним относятся описание формальных в заголовке процедуры (п. 11.1), описания статических ситуаций в заголовке структурного предложения (п. 10.1) и описание параметра цикла в заголовке цикла (п. 9.4). Область действия описаний указывается в последующих разделах при изложении семантики этих конструкций. Общие для всех случаев ограничения состоят в следующем:

— внутри одной и той же цепочки описаний один и тот же идентификатор не может быть описан дважды;

— описание идентификатора должно предшествовать его использованию.

⟨описание⟩ ::=
⟨описание простых констант⟩ | ⟨описание текстов⟩
| ⟨описание простых переменных⟩ | ⟨описание типа⟩
| ⟨описание статических ситуаций⟩ | ⟨описание полей⟩
| ⟨описание меток⟩

5.1. Простые константы. ⟨Описание констант⟩ вводит или доопределяет одну или несколько констант. Имеется три вида описания: полное описание константы, задаваемое с помощью ⟨полной идентификации⟩, предварительное описание, задаваемое с помощью ⟨предварительной идентификации⟩, и доопределение константы, задаваемое с помощью ⟨идентификации доопределения⟩.

Предварительное описание и доопределение используются, как правило, в следующих случаях:

— структурированное изложение текста программы, при котором сначала даются предварительные описания всех констант, а затем их конкретизация путем доопределения;

— описание интерфейса модуля, в котором дается предварительное описание его элементов с последующим их доопределением в теле модуля;

— описание взаимозависимых констант, когда необходимо удовлетворить условию, требующему, чтобы описание идентификатора предшествовало его использованию;

— описание элементов структуры, массива и области.

Описание, начинающееся со служебного слова **процедура**, является сокращенной формой распространенных случаев описания

константы, значением которой является процедура (см. п. 11.1).

⟨описание простых констант⟩ ::= ⟨описание констант⟩

⟨описание констант⟩ ::=
    конст ⟨список идентификации констант⟩
  | **процедура** ⟨список идентификации процедур⟩

⟨идентификация константы⟩ ::=
    ⟨полная идентификация⟩ | ⟨предварительная идентификация⟩
  | ⟨идентификация доопределения⟩

⟨полная идентификация⟩ ::=
    ⟨идентификатор⟩ {#⟨вычисление типа⟩} = ⟨выражение⟩
  | ⟨идентификатор⟩ : ⟨вычисление типа⟩ = ⟨доопределение объекта⟩

⟨предварительная идентификация⟩ ::=
    ⟨идентификатор⟩
  | {⟨идентификатор⟩# , } . . .
    ⟨идентификатор⟩#⟨вычисление типа⟩
  | {⟨идентификатор⟩ : , } . . .
    ⟨идентификатор⟩ : ⟨вычисление типа⟩

⟨идентификация доопределения⟩ ::=
    ⟨идентификатор⟩ = ⟨выражение⟩
  | ⟨идентификатор⟩ = ⟨доопределение объекта⟩

Полное и предварительное описание. Полное описание имеет вид:

конст *x* = *e* **или**
конст *x*#*вт* = *e* **или**
конст *x* : *вт* = ДО

где *x* — ⟨идентификатор⟩, *e* — ⟨выражение⟩, *вт* — ⟨вычисление типа⟩, ДО — ⟨доопределение объекта⟩.

В первых двух случаях вводятся отдельно размещенные константы, а в третьем случае — встроенная константа. Предварительное описание отличается тем, что опущена правая часть описания.

При выполнении ⟨полной идентификации⟩ или ⟨предварительной идентификации⟩ создается новая константа. Обозначением константы является ⟨идентификатор⟩, заданный в описании. Если указано ⟨вычисление типа⟩, то его результат становится типом константы, а иначе константа имеет динамический тип. Дальнейшее выполнение зависит от того, является ли константа отдельно размещенной или встроенной константой.

Отдельно размещенные константы. Если указано ⟨выражение⟩, то его значение становится значением константы. В противном случае значением константы становится пустое значение.

Встроенные константы. В случае констант составного типа создается объект заданного типа. В случае процедурного

типа создается процедура с пустым телом. Этот объект становится значением константы. Если указано ⟨доопределение объекта⟩, то производится доопределение элементов составного объекта или доопределение тела процедуры.

Групповое предварительное описание констант имеет вид:

конст *х*1#, *х*2#, . . ., *хк*#*вт*; или
конст *х*1;, *х*2:, . . ., *хк* : *вт*;

где *х*1, *х*2, . . ., *хк* — ⟨идентификаторы⟩, *вт* — ⟨вычисление типа⟩. Выполнение такого описания состоит в том, что один раз выполняется ⟨вычисление типа⟩, и вводится группа констант (отдельно размещенных или встроенных) данного типа, обозначаемых перечисленными идентификаторами.

Доопределение константы должно входить либо в ту же цепочку описаний, что и предварительное описание этой константы, либо в состав ⟨доопределения объекта⟩, элементом которого является данная константа. Исключение составляет случай раздельной трансляции, описанный в п. 12.1.

Доопределение отдельно размещенной константы имеет вид:

конст *х* = *е*

а доопределение встроенной константы имеет вид

конст *х* = ДО.

Идентификатор *х* обозначает доопределяемую константу. Выполнение доопределения производится следующим образом.

Отдельно размещенная константа. Результат ⟨выражения⟩ становится значением константы.

Встроенная константа. Выполняется доопределение элементов составного объекта или тела процедурного объекта, являющегося значением константы.

Если в правой части ⟨описания константы⟩ находится выражение или доопределение объекта статического класса, то данная константа является константой статического класса.

В правой части отдельно размещенной константы нельзя рекурсивно использовать ⟨идентификатор⟩ данной константы. Выполнению ⟨идентификатора⟩ отдельно размещенной константы должно предшествовать ее полное описание или доопределение. Эти ограничения не распространяются на случай встроенных констант.

5.2. Простые переменные. ⟨Описание простых переменных⟩ вводит одну или несколько переменных и связывает с ними идентификаторы.

⟨описание простых переменных⟩ ::= ⟨описание переменных⟩
⟨описание переменных⟩ ::=
перем ⟨список идентификации переменных⟩

| ⟨простой формат⟩ ⟨список идентификации переменных⟩
⟨идентификация переменной⟩ ::=
⟨полная идентификация переменной⟩
| ⟨предварительная идентификация переменной⟩
⟨предварительная идентификация переменной⟩ ::=
⟨идентификатор⟩
| {⟨идентификатор⟩#,} . . .
⟨идентификатор⟩ # ⟨вычисление типа⟩
⟨полная идентификация переменной⟩ ::=
⟨идентификатор⟩ {#⟨вычисление типа⟩} := ⟨выражение⟩
| ⟨идентификатор⟩ = ⟨идентификатор⟩

В результате выполнения ⟨идентификации переменной⟩ создается новая переменная, обозначением которой является заданный ⟨идентификатор⟩. Если указано ⟨вычисление типа⟩, то результат его выполнения становится типом переменной, а иначе создается переменная динамического типа.

Если для описания переменной используется ⟨простой формат⟩, то тем самым задается переменная динамического типа. ⟨Вычисление типа⟩ в этом случае не указывается. Особенности присваивания таким переменным описаны в § 8.

⟨Идентификация переменной⟩ может содержать ⟨выражение⟩, значение которого задает начальное значение переменной. В этом ⟨выражении⟩ нельзя использовать ⟨идентификатор⟩ из левой части ⟨идентификации переменной⟩.

■ По умолчанию переменная формата ф32 или ф64 инициализируется пустым объектом соответствующего формата, а переменная формата ф128 — пустым объектом формата ф64 *). ⟨Идентификация переменной⟩ выполняется в следующем порядке. Сначала вычисляется значение ⟨выражения⟩, затем для переменной отводится область памяти заданного формата и осуществляется запись начального значения. ■

5.3. Контекстная приставка. Последовательность локальных (или скрытых) описаний можно вводить с помощью контекстной приставки ⟨контекст⟩. Если контекстная приставка находится в начале ⟨типа модуля⟩ (⟨тела модуля⟩, ⟨доопределения процедуры⟩, ⟨закрытого предложения⟩), то областью действия ее ⟨описаний⟩ являются данный ⟨тип модуля⟩ и соответствующее ⟨тело модуля⟩ (⟨тело модуля⟩, ⟨тело процедуры⟩, ⟨закрытое предложение⟩). Если в контекстной приставке указано служебное слово **огр**, то в указанных конструкциях действуют только ⟨описания⟩, вводи-

*) Точнее, в каждое из двух слов, отведенных для переменной, помещается пустой объект формата ф64.

мые в контекстной приставке, а также описания стандартного контекста.

⟨контекст⟩ ::= **контекст** {**огр**} ⟨секция контекста⟩
⟨секция контекста⟩ ::=
    ⟨последовательность описаний или включений⟩
⟨описание или включение⟩ ::=
    ⟨описание⟩ | ⟨включение в контекст⟩
⟨включение в контекст⟩ ::= ⟨имя модуля⟩
⟨имя модуля⟩ ::=
    ⟨идентификатор⟩ | ⟨элемент модуля⟩ | ⟨внешнее имя⟩

⟨Включение в контекст⟩ применяется для того, чтобы ввести в текущий контекст идентификаторы интерфейсных элементов модуля, обозначенного с помощью ⟨имени модуля⟩. Это позволяет обозначать элементы непосредственно с помощью идентификаторов, не прибегая к точечной нотации конструкции ⟨элемент модуля⟩. Кроме того, ⟨включение в контекст⟩ позволяет объединять интерфейсы нескольких модулей в единый интерфейс. А именно, если модуль А включается в состав ⟨секции контекста⟩, задающей интерфейсную часть модуля Б, то интерфейсные элементы модуля А (и их идентификаторы) включаются в состав интерфейсных элементов модуля Б.

Значением конструкции ⟨имя модуля⟩ может быть:

— модуль, являющийся значением константы модульного типа, обозначенной с помощью ⟨идентификатора⟩ или ⟨элемента модуля⟩;

— модуль, являющийся программой-объектом. В этом случае ⟨внешнее имя⟩ должно обозначать указатель, установленный на элемент справочника, ссылающийся на данную программу-объект, или ссылающийся на файл объектного кода, содержащий данную программу-объект (см. § 12).

Описания, включаемые в ⟨секцию контекста⟩ опосредованно с помощью ⟨включения в контекст⟩, имеют такую же область действия, как ⟨описания⟩, непосредственно заданные в этой же ⟨секции контекста⟩. Если ⟨включение в контекст⟩ приводит к конфликту имен, то для обозначения элементов модуля, включенного в контекст, следует применять конструкцию ⟨элемент модуля⟩.

5.4. Доопределение модуля. ⟨Доопределение модуля⟩ используется в правой части ⟨описания константы⟩ и в ⟨генераторе объекта⟩.

⟨доопределение модуля⟩ ::=
      ⟨тело модуля⟩
   | (⟨выражение⟩ {, ⟨список выражений⟩ })
⟨тело модуля⟩ ::=
      ⟨контекст⟩
      начало ⟨последовательность описаний⟩

{ ; ⟨последовательность операторов⟩ }
конец

В ⟨теле модуля⟩ действуют описания скрытых и интерфейсных элементов доопределяемого объекта. Компоненты конструкции ⟨тело модуля⟩ имеют следующее назначение: контекстная приставка ⟨контекст⟩ задает скрытую часть модуля, а ⟨последовательность описаний⟩ задает доопределение интерфейсных элементов модуля, т. е. доопределения констант (в том числе констант типа *тип*), предварительные описания которых даны в ⟨типе модуля⟩. Контекстная приставка содержит доопределение скрытых элементов, предварительные описания которых были даны в ⟨типе модуля⟩, а также вводит новые скрытые элементы. Областью действия этих новых описаний является ⟨тело модуля⟩.

Выполнение ⟨тела модуля⟩ состоит в том, что выполняются входящие в его состав ⟨описания⟩, ⟨включения в контекст⟩ и ⟨последовательность операторов⟩. В результате создается скрытая часть модуля, имеющая такое же время жизни, как и доопределяемый объект.

Вторая альтернатива правила описана в § 11.

5.5. Доопределение процедуры. ⟨Доопределение процедуры⟩ используется в правой части ⟨описания константы⟩ и в ⟨генераторе объекта⟩.

⟨доопределение процедуры⟩ ::=
{ ⟨контекст⟩ } ⟨текст процедуры⟩
| (⟨выражение⟩ { , ⟨список выражений⟩ })

С помощью контекстной приставки ⟨контекст⟩ можно задать константы и переменные, обладающие двумя свойствами. Во-первых, они, будучи созданными при создании процедурного объекта, существуют в течение всего времени жизни процедурного объекта и сохраняют свои значения между вызовами данной процедуры (аналог «собственных величин» Алгола-60); во-вторых, эти константы и переменные доступны только в ⟨тексте процедуры⟩.

Выполнение первой альтернативы правила состоит в следующем. Выполняются ⟨описания и включения⟩ из контекстной приставки, в результате чего создается модуль. Областью действия этих описаний является ⟨текст процедуры⟩. Затем создается процедурный объект, тело которого является ⟨текстом процедуры⟩. Тело вновь созданного процедурного объекта становится телом доопределяемого объекта.

Если доопределение процедуры используется для определения константы, то созданные модуль и процедурный объект являются локальными, а иначе (т. е. ⟨доопределение процедуры⟩ использует-

ся в (генераторе)) время жизни объекта определяется в соответствии с правилами, описанными в § 4.

Вторая альтернатива описана в § 11.

6. Обозначения элементов составных объектов

В этом разделе описываются конструкции, предназначенные для обозначения элемента массива, элемента структуры, элемента модуля, а также подмассива одномерного массива. Например, элемент массива обозначается с помощью конструкции ⟨элемент массива⟩, которая имеет вид:

⟨первичное⟩ [⟨список выражений⟩]

⟨Первичное⟩ задает массив, а ⟨выражения⟩ определяют индексы нужного элемента. Например, *x* [*i*] обозначает *i*-й элемент вектора *x* (нумерация элементов начинается от 0). Число индексов должно быть равно числу измерений массива.

П о д м а с с и в — это подмножество элементов вектора. Часть вектора (подвектор) обозначается с помощью конструкции ⟨подмассив⟩, которая имеет вид:

⟨первичное⟩ [⟨диапазон⟩]

где ⟨первичное⟩ задает исходный вектор, а ⟨диапазон⟩ определяет подмножество. Например, *x* [10 : 20] обозначает часть исходного вектора от элемента с номером 10 до элемента с номером 29, т. е. 20 элементов. Подвектор является вектором в том смысле, что его можно использовать в тех же конструкциях, что и обычный вектор.

6.1. Элемент массива. Конструкция, используемая для обозначения элемента массива, имеет вид:

⟨элемент массива⟩ ::=
⟨первичное⟩ [{⟨формат⟩} ⟨список выражений⟩]

Если ⟨первичное⟩ имеет динамический тип, то различаются два случая: (1) в скобках указан один индекс. В этом случае значением ⟨первичного⟩ должен быть вектор типа *ствект*; (2) в скобках указано более одного индекса. В этом случае значением ⟨первичного⟩ должен быть массив типа *стам* (*к*), где: *к* — число индексов. В перечисленных случаях конструкция ⟨элемент массива⟩ имеет динамический тип.

Если тип ⟨первичного⟩ отличен от динамического типа, но является типом *вм*1, то в скобках должен быть указан один индекс; тип ⟨элемента массива⟩ является динамическим типом. Иначе ⟨первичное⟩ должно иметь тип массива, состоящего из элементов некоторого типа *тх*. Типом конструкции ⟨элемент массива⟩ является тип *тх*.

Использование ⟨формата⟩ в конструкции ⟨элемент массива⟩ описано в гл. 5.

■ *Диапазон индексов.* Индекс по каждому измерению является целым числом или набором и должен заключаться в диапазоне, разрешенном для данного измерения. Если индексы таковы, что смещение обозначаемого элемента относительно начала массива больше общего количества элементов массива, то возникает ситуация *границамассива.* ■

6.2. Элемент структуры. Конструкция, используемая для обозначения элемента структуры, имеет вид:

⟨элемент структуры⟩ ::= ⟨первичное⟩ . ⟨идентификатор⟩

Тип ⟨первичного⟩ должен быть типом структуры. ⟨Идентификатор⟩ обозначает элемент структуры (в том числе параметр типа). Типом данной конструкции является тип обозначаемого элемента.

6.3. Элемент модуля. Конструкция, используемая для обозначения элемента модуля, имеет вид:

⟨элемент модуля⟩ ::=
⟨идентификатор⟩ . ⟨идентификатор⟩
| ⟨элемент модуля⟩ . ⟨идентификатор⟩

⟨Элемент модуля⟩ обозначает элемент, ⟨идентификатор⟩ которого указан после символа «.». Часть конструкции, предшествующая символу «.», должна обозначать константу модульного типа. Типом данной конструкции является тип обозначаемого элемента.

6.4. Подмассив. Конструкция ⟨подмассив⟩ выделяет часть вектора. Значением конструкции является вектор, элементами которого являются элементы выделенной части исходного вектора.

⟨подмассив⟩ ::= ⟨первичное⟩ [{⟨формат⟩} диапазон⟩]
⟨диапазон⟩ ::= {⟨выражение⟩} : {⟨выражение⟩}

Выделяемая часть вектора определяется значениями ⟨выражений⟩. Значение первого ⟨выражения⟩ задает номер начального элемента. Значение второго ⟨выражения⟩ задает количество элементов. Элементы результирующего вектора нумеруются от 0 до $к$—1, где $к$ — количество элементов.

В случае, если одно или оба выражения опущены, их значения выбираются по умолчанию. В качестве начального индекса диапазона выбирается индекс 0. Длина диапазона по умолчанию полагается равной количеству элементов от начального элемента диапазона до конца вектора.

Если ⟨первичное⟩ имеет динамический тип, то его значением должен быть вектор типа *ствект*. В этом случае подмассив имеет динамический тип. В противном случае типом ⟨первичного⟩ должен быть тип вектора, обозначаемый ниже — *тв*. Если *тв* — параметри-

зованный тип, то тип результирующего вектора является связанным подтипом *тв* (*к*), иначе, если *тв* — связанный подтип некоторого типа *тб*, то тип результирующего вектора является связанным подтипом *тб* (*к*). В обоих случаях результирующий тип является статически связанным типом, если ⟨выражение⟩, задающее количество элементов подмассива, представляет собой выражение статического класса.

Использование ⟨формата⟩ в конструкции ⟨подмассив⟩ описано в гл. 5.

■ Если значения выражений таковы, что выделяемая часть выходит за пределы исходного вектора, возникает ситуация границамассива. ■

7. Выражения

Правило для ⟨выражения⟩ объединяет конструкции, которые в результате выполнения выдают значение. Такими конструкциями являются ⟨формулы⟩, представляющие собой традиционную алгебраическую запись правил вычисления значения, ⟨присваивание⟩ переменной, выдающее значение правой части, и ⟨операции над массивами⟩.

⟨выражение⟩ ::=
⟨формула⟩ | ⟨присваивание⟩ | ⟨операции над массивами⟩

7.1. Формула. Формула задает правило для вычисления значения.

⟨формула⟩ ::=
⟨логическая сумма⟩ {**экв** ⟨логическая сумма⟩}. . .
⟨логическая сумма⟩ ::=
⟨логическое произведение⟩ {**или** ⟨логическое произведение⟩}. . .
⟨логическое произведение⟩ ::= ⟨отношение⟩ {**и** ⟨отношение⟩}. . .
⟨отношение⟩ ::=
⟨сумма⟩ { ⟨операция отношения⟩ ⟨сумма⟩ } . . .
| ⟨сравнение строк⟩
⟨сумма⟩ ::=
⟨произведение⟩ { ⟨операция группы сложения⟩
⟨произведение⟩ }. . .
⟨произведение⟩ ::=
⟨сомножитель⟩ {⟨операция группы умножения⟩
⟨сомножитель⟩}. . .
⟨сомножитель⟩ ::=
⟨одноместная формула⟩ {⟨операция степени⟩
⟨одноместная формула⟩} . . .
⟨одноместная формула⟩ ::=
{⟨одноместная операция⟩} ⟨первичное⟩
| ⟨формирование указателя⟩

⟨первичное⟩ ::=

⟨изображение⟩ | ⟨идентификатор⟩
| ⟨элемент структуры⟩ | ⟨элемент модуля⟩
| ⟨элемент массива⟩ | ⟨подмассив⟩
| ⟨квалификация⟩ | ⟨вызов⟩
| ⟨генератор⟩ | ⟨закрытое выражение⟩
| ⟨формирование паспорта⟩ | ⟨формирование⟩
| ⟨форматный обмен⟩ | ⟨признак⟩
| ⟨подстановка текста⟩ | ⟨запрос атрибута⟩
| ⟨модификация атрибута⟩ | ⟨внешнее имя⟩
| ⟨двоичный обмен⟩ | ⟨поле значения⟩
| ⟨указуемая переменная⟩

⟨изображение⟩ ::=

⟨число⟩ | ⟨набор⟩
| ⟨статический вектор⟩ | ⟨текст процедуры⟩
| ⟨статический файл⟩ | ⟨статический справочник⟩
| ⟨пустой объект⟩

⟨Изображения⟩ явно задают объекты и их информационное содержимое. Они служат для обозначения чисел, наборов, статических массивов, процедур, статических файлов, статических справочников и пустых объектов.

Значением ⟨первичного⟩ может быть:

1. Объект, заданный изображением (число, набор, строка и т. д.)
2. Значение переменной или константы, с которой ⟨идентификатор⟩ связан описанием. Сюда относятся значения простых переменных и формальных параметров, а также значения констант. Типом ⟨идентификатора⟩ является тип обозначаемой им переменной или константы.
3. Значение элемента структуры, элемента модуля, элемента массива.
4. Вектор являющийся частью другого вектора.
5. Объект, квалифицированный типом, заданным в ⟨квалификации⟩.
6. Результат ⟨вызова⟩ функции.
7. Объект, динамически созданный с помощью ⟨генератора⟩.
8. Результат выполнения ⟨закрытого выражения⟩.

Остальные случаи разобраны в § 5, 13, 14, 15, 17, 18 этой главы, § 3, 7 гл. 3 и § 1, 2, 4 гл. 5.

Типом ⟨первичного⟩ является тип предложения, которым представлено это ⟨первичное⟩. Последовательность выполнения операций зависит от соотношения их приоритетов: сначала выполняются более приоритетные операции, а затем менее приоритетные. Порядок выполнения последовательности операций одинакового приоритета может зависеть от реализации.

Ниже описываются соотношение приоритетов операций, смысл операций, типы операндов и тип результата. Некоторые операции определены для нескольких типов операндов. В этом случае в описании операции дается список допустимых типов. Если операнды двуместной операции имеют разные типы, то они приводятся к единому типу. Этот тип и указывается в описании операции в графе «тип операндов». Случаи, когда требуются операнды разных типов, оговорены особо. Если тип ⟨формулы⟩, образуемой операцией, зависит от типа операндов, то указывается соответствующий список типов ⟨формулы⟩. Ниже приняты следующие сокращения для обозначения типов: *Ц — цел*, ДЦ — *дцел*, ДЦН — *дцел.* или *наб*, КВ — *квещ*, В — *вещ*, *ДВ — двещ*, КЧ — *кчис*, Ч — *чис*, УНЦ — *унцел*, УНЧ — *унчис.*

■ В процессе выполнения операции может быть обнаружена ошибка в исходных данных или может возникнуть переполнение результата. Если тип значения операнда динамического типа не соответствует тому, который предписывается операцией (с учетом описанной ниже балансировки типов числовых операндов), возникает ситуация *неверныйоперанд.* Остальные случаи описаны в семантике соответствующих операций. ■

7.2. Приоритет операций. Все операции разбиты по приоритетам на восемь групп. Логические двухместные операции имеют низший приоритет: **экв** — первый, **или** — второй, **и** — третий. Ниже приведены синтаксические правила для остальных операций. Приоритет этих операций выше, чем у логических. Правила расположены в порядке возрастания приоритетов.

⟨операция отношения⟩ ::=
⟨операция сравнения⟩ | **<=> среди**
⟨операция сравнения⟩ ::= **< > | = | >| > = | < | <=**
⟨операция группы сложения⟩ ::= **+ | — | +: | —:**
⟨операция группы умножения⟩ ::=*** | умнд |* : | /: | остат**
⟨операция степени⟩ ::= ******
⟨одноместная операция⟩ ::=

целокр	**целобр**	**вещокр**	**вещобр**
ф32окр	**ф32обр**	**ф64окр**	**ф64обр**
цел64окр	**цел64обр**	**в128**	**целзн**
стнаб	**млнаб**	**естьнабор**	**естьпусто**
естьцел	**естьвещ**	**естьф32**	**естьф64**
естьф128	**не**	**пче**	**перв1**
знак	**длина**	**адрес**	**тип**
дес	**бит**	**тег**	

Операции ⟨сравнения строк⟩ имеют такой же приоритет, как и ⟨операции отношения⟩.

Пример. Выражение

*x—y > омега * сум * cos (y+3 * в) ** 2/7.394—8/c[i]—a ** 3*
эквивалентно выражению

*(x—y)>(((((омега * сум) * (cos (y+ (3 * в)) ** 2))/7.394)—*
*— (8/c [i]))— (a ** 3)).*

7.3. Логические операции.

Знак операции	Операция	Тип операндов	Тип формулы
и	конъюнкция	*наб(к), наб*	*наб(к), наб*
или	дизъюнкция	*наб(к), наб*	*наб(к), наб*
экв	эквивалентность	*наб(к), наб*	*наб(к), наб*
не	обращение	*наб(к), наб*	*наб(к), наб*

Если операнды имеют статически связанные типы *наб* (*к*) и *наб* (*п*), причем *к>п*, то тип второго операнда и тип его значения приводится к типу *наб* (*к*). Допускаются операнды статически связанных типов *тва* (*к*) и *твс* (*п*), где *тва* и *твс* — типы векторов элементов формата ф1, причем *к* и *п* не превышает 64. Такие операнды обрабатываются аналогично операндам типов *наб* (*к*) и *наб* (*п*).

В остальных случаях (один или оба операнда имеют тип *наб* или динамически связанный подтип типа *наб*) операнды приводятся к типу *наб*. Операнд *е*, имеющий динамический тип, эквивалентен операнду *е#наб*.

Операция выполняется над наборами равной длины. Если значение одного операнда имеет тип *наб* (*к*), а другого — *наб* (*п*), причем *к>п*, то предварительно производится выравнивание наборов путем увеличения длины второго операнда:

ген *наб(к)* **иниц** ([*к — п* : *п*] : *x*, : 1''0'')

где *x* — значение второго операнда. Операция проц. ·тся поэлементно над первым операндом и вновь сформированным набором. Результат имеет тип *наб* (*к*).

7.4. Операции отношения.

Знак операции *)	Операция	Тип операндов	Тип формулы
= <>	сравнение	Ц, ДЦ, КВ, В, ДВ КЧ, Ч, УНЧ, УНЦ	*лог*
> >=			
< <=			

*) Двухсимвольные обозначения имеют следующий смысл: <> — не равно, >= — больше либо равно, <= — меньше либо равно.

В случае наборов операция сравнения осуществляет поэлементное сравнение операндов. Если необходимо, предварительно производится выравнивание наборов.

■ *Балансировка операндов.* Операции сравнения и приведенные ниже арифметические операции осуществляют предварительную балансировку исходных операндов. Балансировка заключается в следующем:

1. Кроме особо оговоренных случаев, набор преобразуется в целое.

2. Если типы операндов различаются, то один операнд преобразуется к типу другого по схеме: целое → вещественное → числовое. В случае преобразования целого в вещественное преобразуется также тип значения операнда.

3. Если форматы значений операндов различаются, то значение меньшего формата преобразуется к большему формату.

4. Операнд *е* динамического типа обрабатывается эквивалентно операнду вида *е#унчис*. В целочисленных операциях такой операнд обрабатывается эквивалентно операнду вида: *е#унцел*.

Ниже приведена сводная таблица балансировки типов. Параметрами таблицы являются исходные типы операндов, на пересечении строки и столбца указан результирующий единый тип операндов. Поскольку таблица является симметричной, приведена только ее половина.

	Ц	ДЦН	КВ	В	ДВ	КЧ	Ч	УНЧ	УНЦ
Ц	Ц	ДЦ	КВ	В	ДВ	КЧ	Ч	УНЧ	УНЦ
ДЦН		ДЦ	В	В	ДВ	Ч	Ч	УНЧ	ДЦ
КВ			КВ	В	ДВ	КВ	В	УНЧ	УНЧ
В				В	ДВ	В	В	УНЧ	В
ДВ					ДВ	ДВ	ДВ	ДВ	ДВ
КЧ						КЧ	Ч	УНЧ	УНЧ
Ч							Ч	УНЧ	Ч
УНЧ								УНЧ	УНЧ
УНЦ									УНЦ

Преобразование набора в целое происходит следующим образом. Если длина набора меньше 64, то его тип преобразуется к типу *наб* (64), как описано в п. 7.3. Затем из полученного набора *х* формируется результирующее целое:

ген *дцел* **иниц** (*зн* : *х*[0], *мант* : *х* [1 : 63])

Если при этом создается отрицательное число, то возникает ситуация *неверныйоперанд*. При необходимости для преобразования та-

кого набора нужно пользоваться специальной операцией целзн (см. гл. 5).

Преобразование целого в нормализованное вещественное происходит с сохранением формата. Если отбрасываемая часть мантиссы операнда отлична от нуля, происходит округление: в младший разряд мантиссы результата заносится 1. ■

Дополнительные операции сравнения.

Знак операции	Операция	Тип операндов	Тип формулы
<=> *x* среди *y*	проверка совпадения проверка принадлежности	любой *x* — *лит* *y* — логический вектор	*лов* *лог*

Результатом операции проверки совпадения является **истина** в случае, если у значений операндов совпадают типы, форматы, величины и в случае наборов — длины.

Результатом операции проверки принадлежности является результат сравнения *y*[*x*] = **истина.**

7.5. Арифметические операции.

Знак операции	Операция	Тип операндов	Тип формулы
+	сложение	Ц, ДЦ, КВ, В, ДВ, КЧ, Ч, УНЧ, УНЦ	Ц, ДЦ, КВ, В, ДВ, КЧ, Ч, УНЧ, УНЦ
—	вычитание	Ц, ДЦ, КВ, В, ДВ, КЧ, Ч, УНЧ, УНЦ,	Ц, ДЦ, КВ, В, ДВ, КЧ, Ч, УНЧ, УНЦ
*	умножение	Ц, ДЦ, КВ, В, ДВ, КЧ, Ч, УНЧ, УНЦ	Ц, ДЦ, КВ, В, ДВ, КЧ, Ч, УНЧ, УНЦ
/	деление	Ц/КВ, ДЦ/В, ДВ, КЧ, Ч, УНЧ, УНЦ	КВ, В, ДВ, КЧ, Ч, УНЧ, УНЦ
умнд	удвоенное умножение	Ц, ДЦ, КВ, В, КЧ, Ч, УНЧ, УНЦ	ДЦ, ДВ, В, ДВ, Ч, ДВ, УНЧ, УНЦ
x ** *y*	возведение в степень	Ц, ДЦ, КВ, В, ДВ, КЧ, Ч, УНЧ, УНЦ	Ц, ДЦ, КВ, В, ДВ, КЧ, Ч, УНЧ, УНЦ
/:	частное деления нацело	Ц, ДЦ, УНЦ	Ц, ДЦ, УНЦ
остат	остаток деления нацело	Ц, ДЦ, УНЦ	Ц, ДЦ, УНЦ

Операция умнд осуществляет умножение с удвоенной точностью. Формат результата в два раза больше формата значения операндов, но не больше ф128.

В операции возведения в положительную целочисленную степень указан только тип операнда x. Операция осуществляется путем многократного умножения значения левого операнда самого на себя.

Операции целочисленного деления связаны соотношением:

$$x = (x\ /:\ y) * y + (x \textbf{ остат } y)$$

Здесь x **остат** y имеет знак x и абсолютную величину меньшую, чем y. Знак $x\ /:\ y$ определяется так же, как в случае обычного деления.

■ В операциях деления /, /:, **остат** возникает ситуация *делениенануль* в случае, если делитель является целым или вещественным нулем.

Вещественный результат арифметической операции представляется в нормализованном виде. Если отбрасываемая часть мантиссы отлична от нуля, происходит округление: в младший разряд мантиссы результата заносится 1.

В случае переполнения вещественного результата возникает ситуация *переполнвещ* (для форматов ф32 и ф64) или ситуация *переполнвещ128* (для формата ф128). В случае отрицательного переполнения порядка результата выдается вещественный нуль.

В операциях +, —, * над операндами типов КЧ, Ч, УНЧ возможен вещественный результат, даже если значениями операндов являются целые числа. Он возникает в том случае, когда абсолютная величина целочисленного результата превышает максимально допустимую для выбранного формата. Если операнды имеют типы Ц, ДЦ, УНЦ, то аналогичные обстоятельства приводят к возникновению ситуации *переполнцел32* (для формата ф32) или *переполнцел64* (для формата ф64). ■

7.6. Операции преобразования типа. Стандартная функция *измтип* (x, *тн*), где x — аргумент типа *цел* или *дцел*, *тн* — подтип типа *наб*, преобразует абсолютную величину x в набор типа *тн*. Результатом преобразования является набор, получаемый путем копирования значения выражения:

ген *наб*(64) **иниц** (0 : 1"0", [1 : 63] : *х.мант*) [64— $к$: $к$],

где $к$ — длина набора типа *тн*. Если аргумент x имеет тип *цел*, то предварительно производится преобразование к типу *дцел*. Другие применения функции *измтип* описаны в § 2 гл. 5.

Одноместные операции преобразования типа определены для числовых типов.

Преобразование в целое с обрубанием производится путем отбрасывания дробной части. Преобразование в целое с округлением осуществляется по следующей формуле:

$$окр\ (x) = sign\ (x) * обр\ (abs\ (x) + 0.5),$$

где: $sign(x)=1$ для $x>0$, 0 для $x=0$, -1 для $x<0$, *обр* (x) — функция преобразования в целое с обрубанием. Таким образом, округление — это преобразование вещественного к ближайшему целому.

Знак операции	Операция	Тип операнда	Тип формулы
целокр	в целое: с округлением	Ц, ДЦН, КВ, В, КЧ, Ч, УНЧ	Ц, ДЦ, Ц, ДЦ, Ц, ДЦ, УНЦ
целобр	с обрубанием	те же	те же
вещокр	в вещественное: с округлением	Ц, ДЦН, КВ, В, КЧ, Ч, УНЧ	КВ, В, КВ, В, КВ, В, УНЧ
вещобр	с обрубанием	те же	те же
ф32окр	в формат ф32: с округлением	Ц, ДЦН, КВ, В, ДВ, КЧ, Ч, УНЧ	Ц, Ц, КВ, КВ, КВ, КЧ, КЧ, КЧ
ф32обр	с обрубанием	те же	те же
ф64окр	в формат ф64: с округлением	Ц, ДЦН, КВ, В, ДВ, КЧ, Ч, УНЧ	ДЦ, ДЦ, В, В, В, Ч, Ч, Ч
ф64обр	с обрубанием	те же	те же
в128	в формат ф128	Ц, ДЦН, КВ, В, ДВ, КЧ, Ч, УНЧ	ДВ, ДВ, ДВ, ДВ, ДВ, ДВ, ДВ, ДВ
цел64окр	в целое ф64: с округлением	Ц, ДЦН, КВ, В, ДВ КЧ, Ч, УНЧ	ДЦ, ДЦ, ДЦ, ДЦ, ДЦ ДЦ, ДЦ, ДЦ
цел64обр	с обрубанием	Ц, ДЦН, КВ, В, ДВ КЧ, Ч, УНЧ	ДЦ, ДЦ, ДЦ, ДЦ, ДЦ ДЦ, ДЦ, ДЦ

Преобразование целого в вещественное того же формата и сокращение формата вещественного могут привести к отбрасыванию части мантиссы. В связи с этим есть два варианта соответствующих операций — с округлением и с обрубанием. Округление состоит в том, что в младший разряд мантиссы результата заносится 1 в случае, если отбрасываемая часть отлична от нуля. Обрубание представляет собой отбрасывание без округления.

■ Преобразование в целое может привести к переполнению результата. В этих случаях возникает ситуация *переполнцел32* (для формата ф32) или ситуация *переполнцел64* (для формата ф64). Преобразование в вещественное может привести к переполнению порядка. Этим случаям соответствует ситуация *переполнвещ* (для форматов ф32 и ф64). ■

7.7. Операции проверки типа и формата. Описываемые ниже операции являются одноместными. Каждая операция проверяет, совпадает ли тип (или формат) значения операнда с тем типом (или форматом), который в ней указан. В случае совпадения выдается **истина**, иначе — **ложь**.

7.8. Стандартная функция проверки типа. Универсальная функция *провтип* (x, m) с результатом типа *лог* выполняет проверку принадлежности типа значения параметра x типу m, который яв-

Знак операции	Операция	Тип операнда	Тип формулы
естьцел	проверка на: тип Ц, ДЦ	любой	*лог*
естьвещ	тип КВ, В, ДВ	любой	*лог*
естьпусто	пустой объект	любой	*лог*
естьнабор	тип *наб*	любой	*лог*
естьф32	проверка: формата ф32	любой	*лог*
естьф64	формата ф64	любой	*лог*
естьф128	формата ф128	любой	*лог*

ляется вторым параметром. Параметр *т* может иметь вид ⟨вычисления типа⟩. Если проверка прошла успешно, то выдается **истина**, а иначе — **ложь**.

7.9. Прочие одноместные операции.

Знак операции	Операция	Тип операнда	Тип формулы
—, абс	изменение знака, абсолютная величина	Ц, ДЦН, КВ, В, ДВ, КЧ, Ч, УНЧ, УНЦ	Ц, ДЦ, КВ, В, ДВ, КЧ, Ч, КНЧ, УНЦ
знак	знак числа	Ц, ДЦН, КВ, В, ДВ, КЧ, Ч, УНЧ, УНЦ	Ц
длина	длина вектора	вектор	ДЦ
тег	тег объекта	объединенный тип	Ц

Операнд *е* динамического типа в операциях изменения знака, **абс**, **знак** обрабатывается эквивалентно операнду вида *е*#*унчис*. Если операнд операции **длина** имеет динамический тип, то его значением должен быть вектор типа *ствект*, а иначе типом операнда должен быть тип вектора.

Результат операции **знак** определяется следующим соотношением:

$$\textbf{знак}\ x = \begin{cases} 1, & \text{если } x > 0, \\ -1, & \text{если } x < 0, \\ 0, & \text{если } x = 0. \end{cases}$$

Операция **длина** выдает длину вектора.

Операнд операции **тег** должен иметь объединенный тип *то*, отличный от динамического типа. Если значение операнда имеет тип *то* (*к*), то результатом операции является значение *к*, имеющее тип параметра выбора типа *то*.

7.10. Выражения и объекты статического класса. Выражение статического класса — это ⟨формула⟩, в которой ⟨первичными⟩ составляющими могут являться:

а) ⟨число⟩, ⟨набор⟩, ⟨пустой объект⟩;

б) ⟨идентификатор⟩ константы статического класса, т. е. константы, в правой части описания которой находится выражение статического класса;

в) конструкция ⟨поле значения⟩, все компоненты которой являются выражениями статического класса;

г) выражение статического класса в круглых скобках;

д) конструкция ⟨формирование⟩, все компоненты которой являются выражениями статического класса *).

Пример. Выражения статического класса в описаниях констант.

```
конст пи = 3.141593,
      градрад = пи/180,
      е = 2.718282,
      екв = е ** 2;
```

Выражения статического класса выполняются во время трансляции.

Статическим составным объектом называется объект, элементами которого являются константы статического класса. Константой статического класса является:

— отдельно размещенная константа, в правой части описания которой находится выражение статического класса, ⟨строка⟩, генератор массива и структуры статического класса, ⟨статический файл⟩, ⟨статический справочник⟩, ⟨внешнее имя⟩;

— отдельно размещенная константа, описание которой задано с помощью служебного слова **процедура.** В ⟨доопределении процедуры⟩ должны соблюдаться следующие ограничения: если ⟨тексту процедуры⟩ предшествует контекстная приставка, то она должна задавать модуль статического класса; значениями операндов процедурной формы доопределения должны быть процедуры доопределения, заданные с помощью ⟨реализации процедуры⟩, и модули статического класса;

— встроенная константа, в правой части описания которой находится выражение статического класса, ⟨строка⟩, ⟨доопределение составного⟩ объекта статического класса, ⟨доопределение процедуры⟩, в котором должны соблюдаться указанные выше ограничения, ⟨доопределение модуля⟩ статического класса, причем если используется процедурная форма доопределения, то должны соблюдаться указанные выше ограничения, за исключением того, что процедура

*) В этом случае ⟨формирование⟩ не должно приводить к созданию указателя (см. гл. 5).

доопределения задается с помощью ⟨реализации модуля⟩, описывающей создание модуля статического класса;

— константа типа *тип*, значение которой отлично от динамически связанного типа;

— препроцессорная константа (описатель поля, текстовой макрос).

7.11. Квалификация. Эта конструкция позволяет произвести динамический контроль типа объекта и придать ⟨первичному⟩ статически определенный тип.

⟨квалификация⟩ ::= ⟨первичное⟩#⟨вычисление типа⟩

Выполнение конструкции состоит в том, что производится проверка (возможно, динамическая) принадлежности типа объекта, являющегося значением ⟨первичного⟩, типу, являющемуся результатом ⟨вычисления типа⟩. Если условие принадлежности не удовлетворяется, возникает ситуация *неверныйоперанд*, иначе значением конструкции является значение ⟨первичного⟩, а типом конструкции — результат ⟨вычисления типа⟩.

8. Переменная и присваивание

Присваивание служит для изменения текущего значения переменной.

8.1. Переменная. Синтаксическое правило для ⟨переменной⟩ объединяет все возможные способы обозначения переменных.

⟨переменная⟩ ::=

⟨идентификатор⟩	⟨элемент массива⟩
⟨элемент структуры⟩	⟨элемент модуля⟩
⟨указуемая переменная⟩	⟨поле переменной⟩
⟨закрытая переменная⟩	⟨подстановка текста⟩

⟨Идентификатор⟩ обозначает простую переменную или формальный параметр, являющийся переменной. ⟨Элемент массива⟩, ⟨элемент структуры⟩, ⟨элемент модуля⟩ обозначает элемент соответствующего составного объекта, являющийся переменной. ⟨Поле переменной⟩ и ⟨указуемая переменная⟩ рассмотрены в гл. 5.

■ Закрытое предложение (точнее — закрытая переменная) также может служить для обозначения переменных. В этом случае предложение строится таким образом, чтобы его выполнение завершалось выполнением какой-либо конструкции, предназначенной для обозначения переменных. Последняя и задает нужную переменную. Например, условное предложение в левой части присваивания:

если *p* **то** *x* **иначе** *y* **все** := 3.14

обозначает в зависимости от значения *p* либо переменную *x*, либо переменную *y*.

Все альтернативы закрытой переменной должны иметь динамический тип. ■

8.2. Присваивание. ⟨Присваивание⟩ осуществляет замену текущего значения переменной на новое значение, вычисляемое в правой части.

⟨присваивание⟩ ::=
⟨переменная⟩ := ⟨выражение⟩
| ⟨целочисленное присваивание⟩

Выполнение ⟨присваивания⟩ состоит в том, что объект, являющийся значением ⟨выражение⟩, становится значением ⟨переменной⟩, обозначение которой задано в левой части ⟨присваивания⟩.

■ В случае присваивания переменной динамического типа имеют место следующие особенности, связанные с преобразованием формата числового объекта: ц

— переменная формата ф64. Если значение ⟨выражения⟩ имеет тип *двещ*, то оно преобразуется к типу *вещ*. В остальных случаях присваивается исходное значение;

— переменная формата ф128. Значение ⟨выражения⟩ присваивается без преобразования;

— переменная формата ф32. Выполнение правой части (*е*) эквивалентно выполнению ⟨выражения⟩ вида *е#унчис*. Значение типа *дцел* (*вещ*, *двещ*) преобразуется к типу *цел* (*квещ*, *квещ*).

Сокращение формата целого от ф64 к ф32 может привести к преобразованию его в вещественное с обрубанием. Это происходит тогда, когда абсолютная величина исходного целого превышает величину максимально допустимого целого формата ф32. Сокращение формата вещественного до формата ф32 или ф64 может привести к возникновению ситуации *переполнвещ*.

В случае, если переменной формата ф32 присваивается набор, последний сначала преобразуется в целое ф64, а затем уже выполняется присваивание с сокращением формата целого.

Остальные случаи присваивания переменным динамического типа (присваивания переменной строчного формата, полю переменной, полю вектора) рассмотрены в гл. 5. ■

⟨Присваивание⟩ может быть не только ⟨оператором⟩, но и ⟨выражением⟩. В последнем случае его значением является значение правой части.

9. Закрытые предложения

⟨Закрытые предложения⟩ служат для структурирования программы. Для этой цели введены четыре разновидности закрытых предложений:

1. Замкнутое предложение, играющее роль структурных скобок для последовательности описаний и предложений.

2. Условное и выбирающее предложения, которые дают возможность выбора альтернатив для выполнения.

3. Цикл, служащий для организации повторяющегося процесса вычислений.

4. Структурное предложение, являющееся средством структурированной передачи управления при обработке ошибок и других особых ситуаций.

Особенность синтаксического описания ⟨Закрытое предложение⟩ может играть роль ⟨оператора⟩, может использоваться в ⟨выражении⟩ и в конструкциях, где требуется ⟨переменная⟩ (например — в левой части ⟨присваивания⟩). Структура ограничителей некоторого конкретного ⟨закрытого предложения⟩ не зависит от синтаксической позиции, в которой оно находится, но есть различие в том, какими предложениями может завершаться его выполнение. Чтобы не размножать число синтаксических правил по числу возможных позиций, используется параметризация некоторых правил, связанных с ⟨закрытыми предложениями⟩. Признаком параметризации является слово «предложение», встречающееся в левой и правой частях правила. Это слово можно рассматривать как формальный параметр правила. Для того чтобы получить действующее правило, надо в обеих частях правила заменить слово «предложение» на одно из трех слов: «оператор», «выражение», «переменная». Например, из приведенного ниже правила для ⟨закрытого предложения⟩ образуются три правила: для ⟨закрытого оператора⟩, для ⟨закрытого выражения⟩ и для ⟨закрытой переменной⟩.

⟨закрытое предложение⟩ ::=
  {блок} {⟨контекст⟩}
⟨небазированное закрытое предложение⟩
⟨небазированный закрытый оператор⟩ ::=
    ⟨замкнутый оператор⟩ | ⟨условный оператор⟩
  | ⟨выбирающий оператор⟩ | ⟨цикл⟩
  | ⟨структурный оператор⟩ | ⟨выбирающее по типу⟩
⟨небазированное закрытое выражение⟩ ::=
    ⟨замкнутое выражение⟩ | ⟨условное выражение⟩
  | ⟨выбирающее выражение⟩ | ⟨структурное выражение⟩
⟨небазированная закрытая переменная⟩ ::=
    ⟨замкнутая переменная⟩ | ⟨условная переменная⟩
  | ⟨выбирающее переменную⟩

■ Базированный блок. Процедура и ⟨закрытое предложение⟩, начинающееся словом **блок**, называются базированными блоками. В начале выполнения базированного блока созда-

ется связанная с ним базированная область памяти (БОП). В БОП отводится память для переменных и констант, создаваемых с помощью ⟨описаний⟩, и в ряде случаев — память для составных объектов. Если слово **блок** отсутствует, то отводимая память располагается в текущей БОП. При этом распределение ячеек ведется так, чтобы обеспечить максимальное наложение памяти, используемой непересекающимися небазированными блоками. ■

⟨Последовательное предложение⟩ является компонентой ⟨замкнутого предложения⟩ и альтернативных предложений (условного, выбирающего, структурного). Оно используется тогда, когда ограничители этих предложений могут играть роль структурных скобок для заключенной между ними последовательности действий.

⟨последовательное предложение⟩ ::=
{⟨описание⟩;} . . . {⟨помеченный оператор⟩;}. . .
⟨помеченное предложение⟩
⟨помеченное предложение⟩ ::= {⟨метка⟩} . . . ⟨предложение⟩
⟨оператор⟩ ::=

⟨присваивание⟩	⟨модификация атрибутов⟩
⟨вызов⟩	⟨генератор внешнего объекта⟩
⟨структурный переход⟩	⟨запуск задачи⟩
⟨операции над массивами⟩	⟨подстановка текста⟩
⟨закрытый оператор⟩	⟨двоичный обмен⟩
⟨форматный обмен⟩	⟨формирование паспорта⟩
⟨переход⟩	

⟨Последовательное предложение⟩ задает прямую последовательность действий. Сначала выполняются ⟨описания⟩, затем ⟨операторы⟩ и в заключение — ⟨предложение⟩. Областью действия описаний является данное ⟨последовательное предложение⟩.

Результат выполнения. Значением (типом) ⟨последовательного выражения⟩ является значение (тип) завершающего ⟨выражения⟩. ⟨Последовательная переменная⟩ обозначает переменную, определяемую в результате выполнения завершающей ⟨переменной⟩. Этот результат, в свою очередь, является результатом объемлющей конструкции (⟨закрытого выражения⟩ или ⟨закрытой переменной⟩).

Если типы всех альтернатив ⟨условного выражения⟩, ⟨выбирающего выражения⟩, ⟨структурного выражения⟩ статически совпадают, то этот тип является типом всего ⟨закрытого выражения⟩. В противном случае считается, что ⟨закрытое выражение⟩ имеет динамический тип.

9.1. Замкнутое предложение. ⟨Замкнутое предложение⟩ представляет собой простейший способ структурирования описаний и предложений. Оно используется в основном для организации бло-

ков с локальными описаниями (в частности, для оформления тела процедуры), а также для изменения порядка вычислений в ⟨формулах⟩. ⟨Замкнутое предложение⟩ представляет собой ⟨последовательное предложение⟩, заключенное в скобки: (и), или **начало** и **конец**.

⟨замкнутое предложение⟩ ::=
((последовательное предложение))
| **начало** ⟨последовательное предложение⟩ **конец**

Пример. Блок с описанием.

начало перем *с*#*чис* := 0;
для *i* **до** *вг* **цикл**
с := *с*+ *f* (*i*) ** 2
повторить
конец

Пример. ⟨Замкнутое предложение⟩ в роли ⟨выражения⟩.
a := (*b*+*c*)**d*

9.2. Условное предложение. Условное предложение позволяет из нескольких альтернатив выбрать и выполнить ту, для которой ⟨условие⟩ принимает значение **истина**.

⟨условное предложение⟩ ::=
если ⟨условие⟩ **то** ⟨последовательное предложение⟩
{ **инес** ⟨условие⟩ **то** ⟨последовательное предложение⟩}. . .
{ **иначе** ⟨последовательное предложение⟩}
все
⟨условие⟩ ::=
{⟨описание⟩;} . . . {⟨помеченный оператор⟩;} . . .
⟨помеченное выражение⟩

При выполнении ⟨условного предложения⟩ последовательно выполняются ⟨условия⟩ до первого, ⟨выражение⟩ которого принимает значение **истина**. Затем выполняется соответствующее ⟨последовательное предложение⟩. Если такого условия не нашлось, но ⟨условное предложение⟩ содержит альтернативу **иначе**, то выполняется ⟨последовательное предложение⟩ из этой альтернативы. Если же ни одно условие не выполнилось и нет альтернативы **иначе**, то ни одно ⟨последовательное предложение⟩ не выполняется. ⟨Условное выражение⟩ и ⟨условная переменная⟩ с необходимостью должны содержать альтернативу **иначе**.

⟨Условие⟩ может содержать ⟨описания⟩. Областью действия каждого такого описания является вся следующая за ним часть ⟨условного предложения⟩, вплоть до закрывающего ограничителя **все**.

Пример.

```
если месяц = 12 и послдень (год, месяц, день) то
   месяц := 1;
   день := 1;
   год := год+1
инес послдень (год, месяц, день) то
   месяц := месяц+1;
   день := 1
иначе день := день + 1
все
```

Пример. Сокращенная запись предыдущего примера.

```
день := если послдень (год, месяц, день) то
          месяц := если месяц = 12 то
                     год := год + 1;
                     1                  % первый
                                        % месяц след. года
                   иначе месяц + 1      % следующий
                                        % месяц
                   все;
          1                             % первый день
                                        % след. месяца
       иначе день + 1                   % следующий день
       все
```

Процедура *послдень* описана в примере п. 9.3.

9.3. Выбирающее предложение. ⟨Выбирающее предложение⟩ позволяет выбрать одну из пронумерованных альтернатив и выполнить ее. Номер выполняемой альтернативы вычисляется в заголовке ⟨выбирающего предложения⟩.

```
⟨выбирающее предложение⟩ ::=
   выбор ⟨вычисление номера⟩ из
   ⟨список размеченных альтернативных предложений⟩
   {иначе ⟨последовательное предложение⟩}
   всевыб
  |выбор ⟨вычисление номера⟩ из
   ⟨список последовательных предложений⟩
   {иначе ⟨последовательное предложение⟩}
   всевыб
⟨вычисление номера⟩ ::=
   {⟨описание⟩;} ... {⟨помеченный оператор⟩;} ...
    ⟨помеченное выражение⟩
⟨размеченное альтернативное предложение⟩ ::=
   {⟨номер⟩:,} ... ⟨номер⟩: ⟨последовательное предложение⟩
⟨номер⟩ ::= ⟨выражение⟩
```

Альтернативы ⟨выбирающего предложения⟩ нумеруются либо в явной форме, либо неявно. В первом случае ⟨выбирающее предложение⟩ строится из ⟨размеченных альтернативных предложений⟩. Каждой альтернативе можно приписать один или несколько ⟨номеров⟩. ⟨Номер⟩ — это выражение статического класса, значением которого может быть целое или набор.

Во втором случае номер в явном виде отсутствует, и считается, что альтернативы пронумерованы от 0 до $n-1$, где n — число альтернатив.

Выполнение ⟨выбирающего предложения⟩ происходит следующим образом. Выполняется ⟨вычисление номера⟩. Значение ⟨выражения⟩ определяет номер альтернативы, которую следует выполнить. Если такая альтернатива существует, то выполняется ее ⟨последовательное предложение⟩. Если же такой альтернативы нет, но есть альтернатива **иначе**, то выполняется последовательное предложение из альтернативы **иначе**. В противном случае ни одно ⟨последовательное предложение⟩ не выполняется. ⟨Выбирающее выражение⟩ и ⟨выбирающее переменную⟩ обязательно должны содержать альтернативу **иначе**.

Примеры.

```
выбор день из
1:, 3:, 7: печать (стр8"дневная смена"),
2:, 4: печать (стр8"вечерняя смена"),
5:, 6: печать (стр8"выходной день")
иначе печать (стр8"неверно задан день")
всевыб
процедура послдень = функция (год, месяц, день)
   день = выбор месяц из
          1:, 3:, 5:,
          7:, 8:, 10:, 12: 31,
          4:, 6:, 9:, 11: 30,
          2: если год остат 4 = 0 и
                год остат 100 < > 0 или
                год остат 400 = 0 то
             29     % високосный год
             иначе 28
             все
          иначе 0 всевыб
приоритет :=
   выбор лексема из
   "или":2, "и":3        % логические операции
   "<":, ">":, "=":, "<=":, ">=":, "< >":4,
                         % операции отношения
   "+":, "—":5,          % операции типа сложения
```

```
"*":, "/": 6,          % операции типа умножения
"**": 7                % возведение в степень
иначе
  ошибка (стр8"нет такой операции");
всевыб
```

Предложение, выбирающее по типу, состоит из альтернатив, помеченных непересекающимися типами. Это предложение позволяет в зависимости от типа значения ⟨выражения⟩, находящегося в заголовке предложения, выбрать и выполнить альтернативу, помеченную этим типом. Альтернатива выполняется в контексте описания константы, значением которой является значение ⟨выражения⟩, а типом — тип этого значения.

⟨выбирающее по типу⟩ ::=
выбтип ⟨выражение⟩ **из**
⟨список размеченных альтернатив типа⟩
{**иначе** ⟨последовательное предложение⟩}
всевыб
⟨размеченная альтернатива типа⟩ ::=
⟨описание альтернативы⟩ : ⟨последовательное предложение⟩
⟨описание альтернативы⟩ ::=
⟨идентификатор⟩ # ⟨вычисление типа⟩
| ⟨идентификатор⟩ # (⟨параметр⟩)
⟨параметр⟩ ::= ⟨выражение⟩

При выполнении ⟨выбирающего по типу⟩ производятся следующие действия. Для каждой из альтернатив в порядке их расположения формируется описание константы

конст $x\#\#m = e\#\#m$

где: x — ⟨идентификатор⟩, заданный в ⟨описании альтернативы⟩; m — либо ⟨вычисление типа⟩, заданное в ⟨описании альтернативы⟩, либо ⟨вычисление типа⟩ вида z (⟨параметр⟩), если тип выражения в заголовке является объединенным типом z, отличным от динамического типа; e — ⟨выражение⟩ из заголовка. Если выполнение квалификации $e\#\#m$ не приводит к возникновению ситуации *неверныйоперанд*, то выполняется соответствующее ⟨последовательное предложение⟩ в контексте описания константы x, и на этом завершается выполнение ⟨выбирающего по типу⟩. В противном случае такое же действие производится для следующей альтернативы. Если успешно завершенной альтернативы нет, выполняется альтернатива **иначе**. Множества, типы которых указаны в ⟨описании альтернатив⟩, не должны пересекаться.

9.4. Цикл. Конструкция ⟨цикл⟩ предназначена для организации повторяющегося процесса выполнения ее ⟨операторов⟩.

⟨цикл⟩ ::=
{⟨заголовок цикла⟩}
цикл
{⟨описание⟩;} . . .
{⟨помеченный оператор⟩;} . . .
⟨помеченный оператор⟩
повторить

Повторяющееся выполнение ⟨операторов⟩ цикла без заголовка может быть прекращено с помощью ⟨структурного перехода⟩, пересекающего границы цикла (и, кроме того, с помощью обычного перехода на метку).

Цикл с заголовком определяет, что ⟨операторы⟩ выполняются столько раз, сколько задано в ⟨заголовке цикла⟩.
⟨заголовок цикла⟩ ::=

для ⟨идентификатор⟩ {**от** ⟨выражение⟩} ⟨компонента-до⟩
{**шаг** ⟨выражение⟩}
| **от** ⟨выражение⟩ ⟨компонента-до⟩
⟨компонента-до⟩ ::= **до** ⟨выражение⟩| **вниздо** ⟨выражение⟩

В ⟨заголовке цикла⟩ задается ⟨идентификатор⟩ параметра цикла. Появление этого идентификатора в заголовке одновременно является и его описанием в пределах данного ⟨цикла⟩. Начальное значение параметра цикла задается ⟨выражением⟩ в компоненте **от**, конечное значение — ⟨выражением⟩ в ⟨компоненте-до⟩, а шаг изменения — статическим ⟨выражением⟩ в компоненте **шаг**. Ограничитель ⟨компоненты-до⟩ задает направление изменения параметра цикла: на каждой итерации его значение увеличивается в случае ограничителя **до** и уменьшается в случае ограничителя **вниздо**.

Семантика выполнения цикла с заголовком отображается следующей программой:

...установить начальное значение *i*...;
до *кц*
цикл
если (*i* — *u*) * *s*>0 **то** кц! **все**;
. . .выполнить операторы цикла. . .;
. . .увеличить (уменьшить) *i* на *k* . . .
повторить

где *i* — параметр цикла (начальное значение задается в заголовке), *u* — конечное значение параметра цикла, *s*=1 для цикла по возрастанию значения параметра и *s*=—1 для цикла по убыванию значения параметра, *k* — шаг изменения параметра цикла.

Начальное и конечное значение параметра цикла вычисляется один раз при входе в цикл. Значения ⟨выражений⟩ должны

быть целыми числами. В ⟨выражениях⟩ ⟨заголовка цикла⟩ нельзя использовать ⟨идентификатор⟩ параметра того же цикла. Если компонента **от** и/или компонента **шаг** опущены, то начальное и/или шаг изменения выбираются по умолчанию. Умолчание для начального значения —0, а для шага —1.

Параметр цикла на каждой итерации обрабатывается как константа типа *цел*. Его значение нельзя изменять с помощью ⟨присваивания⟩, его нельзя использовать в качестве параметра, передаваемого именем, и в качестве объекта при формировании указателя (см. п. 2, гл. 5).

Идентификатор параметра цикла действует только внутри цикла. Однако, если выполнение цикла прекращается ⟨структурным переходом⟩, то текущее значение параметра можно передать в качестве фактического параметра реакции. ■

Пример. Найти значение x в векторе $a[0 : n+1]$

```
начало
   a[n] := x;
   ix := до найден
         для i до n
         цикл
           если a[i] = x то найден! (i) все
         повторить
         при найден (к): если к = n то —1 иначе к все
         всесит
конец
```

Компоненту **для** можно опускать в том случае, если в ⟨операторах⟩ цикла параметр цикла не используется. ■

⟨Описания⟩ из тела ⟨цикла⟩ выполняются один раз при входе в цикл. Областью действия каждого такого описания является вся последующая часть ⟨цикла⟩, вплоть до закрывающего ограничителя **повторить**.

Пример. Умножение матриц a и $в$ размера $n \times n$.

```
для i до n—1
цикл перем cij##чис;      % промежуточные вычисления проводятся
                          % с двойной точностью
   для j до n—1
   цикл
         cij := 0;
         для к до n—1
         цикл
           cij := cij + a [i, к] умнд в [к, j]
         повторить;
```

```
        c[i, j] := вещ64окр cij % округление результата
    повторить
повторить
```

10. Обработка ситуаций

Семантика конструкций описывает нормальный процесс их выполнения. В частности, под термином «завершение» понимается нормальное окончание выполнения конструкции. В отличие от этого в данном разделе описываются средства языка, предназначенные для программирования алгоритмов обработки ошибок и других особых обстоятельств, при которых прекращается процесс нормального выполнения предложений программы. Эти обстоятельства объединяются под общим названием «ситуации». О возникновении ситуации сигнализирует конструкция ⟨структурный переход⟩, которая (без учета возможного списка параметров) имеет вид:

⟨первичное⟩!

⟨Первичное⟩ задает возникшую ситуацию. ⟨Структурный переход⟩ прекращает выполнение ⟨закрытого предложения⟩, управляющего данной ситуацией. Говорят, что ⟨закрытое предложение⟩ управляет ситуацией *s* в том случае, если оно входит в состав ⟨структурного предложения⟩, в заголовке которого указана данная ситуация *s*. ⟨Структурное предложение⟩ имеет вид:

до ⟨список определения ситуаций⟩
⟨закрытое предложение⟩
{приставка альтернативных предложений}

Например, цикл, входящий в состав ⟨структурного предложения⟩,

```
до найденнуль
цикл
    читф(вхфайл, имя x); % ввести следующее число
    если x = 0 то найденнуль! все;
    сумма := сумма + x
повторить
```

управляет ситуацией *найденнуль*. При возникновении этой ситуации выполнение цикла прекращается.

Прекращение выполнения, в отличие от завершения, означает, что пропускаются все действия от точки возникновения ситуации до конца соответствующего ⟨закрытого предложения⟩ (в приведенном примере оператор присваивания на последней итерации не выполняется). В заключение выполняется реакция на ситуацию,

если она была предусмотрена в «приставке альтернативных предложений».

10.1. Структурное предложение. ⟨Структурное предложение⟩ служит для того, чтобы выделить ⟨закрытое предложение⟩, управляющее определенными ситуациями. Таких ситуаций может быть несколько. Все они определяются в заголовке ⟨структурного предложения⟩. После заголовка следует выделяемое ⟨закрытое предложение⟩ ::=

⟨структурное предложение⟩ ::=
до ⟨список определений ситуаций⟩
⟨закрытое предложение⟩
{**при** ⟨список альтернативных предложений⟩ **всесит**}

В конструкции может быть задан ⟨список альтернативных предложений⟩, в котором указываются реакции на ситуации, определенные в заголовке. Если список полностью опущен или же в нем указаны реакции не для всех ситуаций, то это означает, что соответствующие реакции выбираются по умолчанию.

■ Если ⟨закрытое предложение⟩, входящее в состав ⟨структурного предложения⟩ с приставкой, в свою очередь, является ⟨структурным предложением⟩, то оно также должно содержать приставку. ■

Выполнение ⟨структурного предложения⟩ завершается либо после нормального завершения его ⟨закрытого предложения⟩, либо после нормального завершения реакции на ситуацию.

Закрытое предложение может управлять как статической, так и динамической ситуацией.

⟨определение ситуации⟩ ::= ⟨идентификатор⟩ | * ⟨идентификатор⟩

Статической ситуацией управляет только одно ⟨закрытое предложение⟩. Связь предложения и ситуации задается одним из двух способов:

— с помощью ⟨определения ситуации⟩ в форме: ⟨идентификатор⟩. Такая форма определения является описанием, которое вводит статическую ситуацию и связывает с ней ⟨идентификатор⟩. Это описание действует в пределах данного ⟨структурного предложения⟩;

■ — с помощью ⟨определения ситуации⟩ в форме: *⟨идентификатор⟩, где ⟨идентификатор⟩ обозначает ситуацию, описанную ранее с помощью предварительного ⟨описания статической ситуации⟩.

⟨описание статических ситуаций⟩ ::=
статсит ⟨список идентификаторов⟩

Предварительное описание предназначено для того, чтобы описание процедур, в которых возникает эта ситуация, могло тексту-

ально предшествовать ее определению в ⟨структурном предложении⟩.

Пример.

```
начало
   статсит s;
   процедура p=проц (. . .) начало. . . s! . . . конец;
   . . .
   до *s
   начало
   . . . p (. . .). . .
   конец
   . . .
конец
```

Предварительное описание вводит статическую ситуацию и связывает с ней ⟨идентификатор⟩. Одно описание может вводить несколько ситуаций. Такое описание имеет обычную область действия. В этой области действия ситуацией может управлять лишь одно ⟨закрытое предложение⟩. ■

Динамической ситуацией, в отличие от статической, могут одновременно управлять несколько ⟨закрытых предложений⟩. В частности, допускается случай, когда в процессе выполнения программы эти предложения оказываются динамически вложенными одно в другое.

Динамическая ситуация вводится не с помощью статического описания, а создается динамически в результате выполнения ⟨генератора ситуации⟩.

⟨генератор ситуации⟩ ::=
ситуация ({⟨список установки атрибутов⟩})

Чтобы в ⟨списке определения ситуаций⟩ указать динамическую ситуацию, надо использовать форму :* ⟨идентификатор⟩, где ⟨идентификатор⟩ обозначает константу или переменную, текущим значением которой является динамическая ситуация.

■ Ситуация имеет два атрибута: *процкр* и *локал*. Значением атрибута *процкр* является процедура конечной реакции на ситуацию. Если этот атрибут опущен, то считается, что конечная реакция не предусмотрена.

Значение атрибута *локал* определяет локализацию ситуации.

В стандартном контексте (см. § 12) описаны константы, значениями которых являются стандартные динамические ситуации, связанные с ошибками вычислений, с ошибками, которые могут возникнуть в процессе взаимодействия с внешними объектами и др. Любое предложение программы может взять на себя управление такой ситуацией. Например:

```
до * ситлкф
цикл
   . . . чит(вхтекст,. . .)
повторить
при
   ситлкф: . . . R . . .
всесит
```

Здесь чтение файла *вхтекст* продолжается до тех пор, пока не встречен логический конец файла (соответствующая ситуация является значением глобальной константы *ситлкф*), после чего выполнение цикла прекращается и предпринимается реакция R. Обстоятельства, при которых возникают стандартные ситуации, описаны в соответствующих разделах семантики. Идентификаторы стандартных ситуаций перечислены в Приложении 6. ■

Реакция на ситуацию. Приставка ⟨структурного предложения⟩ состоит из ⟨альтернативных предложений⟩, описывающих реакции на ситуации, которыми управляет ⟨закрытое предложение⟩, входящее в состав данного ⟨структурного предложения⟩.

⟨альтернативное предложение⟩ ::=
{⟨идентификатор⟩:,} . . . ⟨идентификатор⟩
{(⟨список предварительных идентификаций⟩)}
: ⟨последовательное предложение⟩

⟨Идентификаторы⟩, находящиеся в начале альтернативы, перечисляют ситуации, для которых предусмотрена данная реакция. Для обозначения статической или динамической ситуации используются те же идентификаторы, что и в заголовке данного ⟨структурного предложения⟩. ⟨Последовательное предложение⟩ задает действия, которые надо выполнить в качестве реакции.

В ⟨альтернативное предложение⟩ могут входить описания формальных, предназначенные для параметризации реакции. Каждый формальный параметр эквивалентен константе, которой в момент инициирования реакции присваивается значение, равное значению соответствующего фактического параметра структурного перехода.

В случае, если для какой-либо ситуации, определенной в заголовке, реакция явно не задана (в частности, если приставка полностью опущена), она выбирается по умолчанию. Умолчание определяется следующим образом:

— в случае ⟨структурного оператора⟩ реакция представляет собой пустое действие;

— в случае ⟨структурного выражения⟩ выдается значение фактического параметра ⟨структурного перехода⟩. Это свойство

можно, в частности, использовать при программировании процедур-функций. Например, выполнение функции

```
функция (. . .)
   до резвычислен
   начало
     . . .
     резвычислен! (e);
     . . .
   конец
```

прекращается при возникновении ситуации *резвычислен*. Результирующим значением является значение выражения *e*.

■ В реакции ⟨структурного предложения⟩ нельзя использовать статическую ситуацию, определенную в его заголовке (так как это приведет к зацикливанию). В отличие от этого динамическую ситуацию можно использовать. В частности, ее можно использовать в ⟨структурном переходе⟩ из реакции. При этом необходимо учитывать, что в момент инициирования реакции восстанавливается смысл, приданный динамическим ситуациям в ⟨структурных предложениях⟩, охватывающих данное (см. п. 10.2). ■

10.2. Структурный переход. ⟨Структурный переход⟩ сигнализирует о возникновении ситуации.

⟨структурный переход⟩ ::= ⟨первичное⟩!
{(⟨список выражений⟩)}

⟨Структурный переход⟩ по ситуации *s*, являющейся значением ⟨первичного⟩, прекращает выполнение группы предложений и процедур, динамически охватывающих ⟨структурный переход⟩. В эту группу входят все предложения и процедуры, начиная с ближайшего, охватывающего ⟨структурный переход⟩, и кончая тем ⟨закрытым предложением⟩, которое управляет ситуацией *s*.

■ Прекращение выполнения блока (в том числе процедуры), как и его завершение, приводит к тому, что локализованные в нем объекты ликвидируются. ■

В заключение выполняется реакция на ситуацию *s*, указанная в том ⟨структурном предложении⟩, куда произошел возврат.

В случае динамической ситуации имеет место следующая особенность. Если ⟨структурный переход⟩ охвачен одновременно несколькими ⟨закрытыми предложениями⟩, управляющими одной и той же ситуацией *s*, то прекращается выполнение только ближайшего из них.

Пример.

```
начало
   конст s=ситуация ( ),
         p = проц (x) начало. . .s!. . .конец;
```

```
. . .
до * s
начало
% блок A
   . . .
   до * s
   начало
   % блок B
      . . .
      p (z);
      . . .
   конец
   при
   s :. . . RB . . .
   всесит;
   . . .
   p (y);
   . . .
конец
при
s : . . . RA . . .
всесит;
. . .
конец
```

Здесь может произойти следующее.

1. Если ситуация s возникает в процессе выполнения вызова $p(z)$, то прекращается выполнение процедуры p и блока B, затем выполняется реакция RB; после этого выполняются операторы, следующие за ⟨структурным предложением⟩ B.

2. Если ситуация s возникает при вызове $p(y)$, то прекращается выполнение процедуры p и блока A, затем выполняется реакция RA; после этого выполняются операторы, следующие за ⟨структурным предложением⟩ A.

■ К о н е ч н а я р е а к ц и я. В процессе выполнения ⟨структурного перехода⟩ по динамической ситуации может оказаться, что в данном независимом процессе (см. п. 11.2) не существует охватывающего ⟨закрытого предложения⟩, управляющего этой ситуацией. В этом случае прекращается выполнение независимого процесса, в котором возникла ситуация. Затем выполняется конечная реакция, заданная при генерации ситуации. Если конечная реакция не предусмотрена, выдается сообщение «не переопределена ситуация». В заключение независимый процесс ликвидируется.

Необходимо учитывать, что к началу выполнения конечной реакции оперативные объекты (массивы, оперативные объекты

связи с внешними объектами) и внешние объекты, локализованные в прекращенном процессе, уже ликвидированы. По этой причине подобные объекты нельзя использовать в конечной реакции. ■

Параметры. Конструкция может содержать ⟨список выражений⟩, значения которых передаются в реакцию (в частности, — в конечную реакцию) в качестве фактических параметров.

Пример. Передача параметров в реакцию.

```
для i от 2 до n
цикл  % сортировка массива a методом простой
      % вставки; значение элемента a[0]
      % является ограничителем при поиске
   перем x#чис;
   x := a[0]:= a[i];
   до найденменьшех
   для j от i—1 вниздо 0
   цикл
     если a[j] <= x то найденменьшех! (j)
     иначе a[j+1]: = a[j] % сдвиг
     все
   повторить
   при найденменьшех(к): a[к + 1]: = x
   всесит
повторить
```

■ Пример. Использование стандартных ситуаций.

```
процедура факториал = функция (конст n#цел) #цел
   до * переполнцел32
   если n <= 1 то 1
   иначе n * факториал (n—1)
   все
   при переполнцел32: максцел
   всесит
```

В этом примере при выполнении целочисленного умножения (см. п. 7.5) может возникнуть стандартная ситуация *переполнцел*32 (переполнение результата операции). В этом случае функция выдает максимальное целое, что приводит к возникновению ситуации *переполнцел*32 в предыдущем поколении активации функции и т. д. В заключение первое поколение активации выдаст максимальное целое. ■

11. Элементы вычислений

В этом параграфе описываются три элемента вычислений: выполнение процедуры, процесс и задача.

11.1. Процедура. Базовой конструкцией языка, предназначенной для использования процедурного механизма, является ⟨текст процедуры⟩, имеющий вид:

⟨спецификация процедуры⟩ ⟨закрытое предложение⟩

Эта конструкция задает собственно содержание процедуры (спецификацию параметров и результата, а также тело процедуры, т. е. описание алгоритма в виде ⟨закрытого предложения⟩), но не связывает процедуру с идентификатором. Значением ⟨текста процедуры⟩ является объект процедурного типа (далее будем говорить не «объект процедурного типа», а просто «процедура»).

Средства работы с процедурой обычные: она может быть присвоена переменной, передана в качестве фактического параметра, процедура может стать значением константы. Например, если дана переменная *р* динамического типа, то в результате ⟨присваивания⟩

р := **проц** (. . .) **начало** . . . **конец**

ее значением становится процедура, изображенная в правой части. Пока процедура является значением *р*, можно считать, что *р* — это идентификатор данной процедуры.

При необходимости можно создать неразрывную связь между процедурой и идентификатором. Это делается путем задания ⟨текста процедуры⟩ в правой части ⟨описания простой константы⟩.

Пример.

```
конст скапр = функция (а, в)
   начало перем сум#цнчцо := вещ128 0;
     для i до (длина а — 1)
     цикл
       сум := сум + а[i] умнд в [i]
     повторить;
     вещ64окр сум
   конец
```

⟨Спецификация процедуры⟩ может начинаться со служебного слова **проц** или **функция**. Первое задает процедуру, в результате выполнения которой значение не выдается, а второе — процедуру-функцию.

⟨Вызов⟩ процедуры имеет вид:

⟨первичное⟩ (...фактические параметры...)

где значение ⟨первичного⟩ — это вызываемая процедура. Например, если значением переменной *p* и константы *скапр* являются процедуры, то их вызов обозначается следующим образом:

p (... фактические параметры ...)
скапр (... фактические параметры ...).

Существует четыре способа передачи параметров: «значением», «именем», «значение подпрограммой», «имя подпрограммой» (см. ниже).

Текст процедуры. Правило для ⟨текста процедуры⟩ имеет следующий вид:

⟨текст процедуры⟩ ::=
⟨спецификация процедуры⟩ ⟨закрытый оператор⟩
| ⟨спецификация процедуры⟩ ⟨закрытое выражение⟩
| ⟨реализация процедуры⟩ | ⟨реализация модуля⟩
⟨спецификация процедуры⟩ ::=
проц ({⟨параметры⟩})
| **функция** ({⟨параметры⟩})
| ⟨тип процедуры⟩
⟨параметры⟩ ::= ⟨последовательность описаний формальных⟩
⟨описание формального⟩ ::=
{⟨простой формат⟩} ⟨список идентификаторов⟩
| ⟨описание базы⟩

⟨Спецификация процедуры⟩ определяет тип процедуры: первая и вторая альтернативы соответствуют стандартному типу *стпроц* (см. выше пример функции *скапр*), а третья альтернатива задает указанный в спецификации программно определяемый тип процедуры. Служебное слово **проц** задает процедуру, в результате выполнения которой значение не выдается. Ее тело представляет собой ⟨закрытый оператор⟩. Такую процедуру нельзя вызвать в ⟨выражении⟩ (точнее, вызов этой процедуры не может являться ⟨первичным⟩).

Служебное слово **функция** задает процедуру-функцию. В результате выполнения функции выдается значение ее ⟨закрытого выражения⟩. Например, выполнение приведенной в предыдущем разделе процедуры *скапр* завершается вычислением формулы **вещ64окр** *сум*, результат которой является значением этой функции. Тип ⟨закрытого выражения⟩ должен статически принадлежать типу тела процедуры-функции (см. п. 3.1). Процедура-функция стандартного типа всегда имеет тело динамического типа. Вызов процедуры-функции может выполнять роль ⟨первичного⟩ и роль ⟨оператора⟩.

Формальные параметры. В ⟨спецификации процедуры⟩ могут быть заданы описания формальных параметров. Эти

описания аналогичны по форме описаниям простых переменных и констант. В процедурах стандартного типа задается только формат параметра, являющегося переменной динамического типа. По умолчанию выбирается формат ф64. Областью действия формальных параметров является ⟨закрытое предложение⟩.

Спецификация процедуры без параметров заканчивается парой скобок: ( ).

■ При вызове процедуры образуется базированный блок, у которого в начале БОП располагаются фактические параметры. Если ⟨закрытое предложение⟩ является небазированным блоком, то память для локальных описаний отводится следом в той же самой БОП. В противном случае для ⟨закрытого предложения⟩ создается новая БОП.

⟨Текст процедуры⟩ стандартного типа с переменным числом параметров может быть организован следующим образом:

проц (. . .; **база** *перемпарам*)
блок . . . *перемпарам* [*к*] . . .

Здесь тело процедуры является базированным блоком; ⟨описанию базы⟩ предшествует описание фиксированной части параметров (если такая есть). При вызове такой процедуры нужно сначала указывать фиксированную часть фактических параметров, а затем переменную часть. Значением *перемпарам* является вектор типа *свкор* (см. § 16); *перемпарам* [*к*] обозначает к-й элемент. Например, ⟨текст процедуры⟩ вывода чисел может иметь такой вид:

конст *печатьчисел* = **проц** (*файл*; **база** *числа*)

```
блок
начало
  конст числаф64 = числа [ф64:]; % преобразование формата
  . . .
  для к до (длина числаф64—1)
  цикл
  . . .числаф64 [к] . . . % обращение к к-му параметру
                          % переменной части
  повторить;
конец
```

В списке фактических параметров этой процедуры на первом месте надо указывать файл; за ним следует список любой длины задающий выводимые числа:

печатьчисел (*файлацпу*, *е1*, *е2*, . . ., *еп*) ■

При вызове процедуры фактические параметры ставятся в соответствие формальным. Способ передачи фактического параметра задается в месте вызова. Он определяет:

1. Вычисляется ли фактический параметр один раз при входе в процедуру, или же каждый раз при обращении к соответствующему формальному параметру.

2. Что ставится в соответствие формальному параметру — переменная, являющаяся фактическим параметром, или ее значение.

Способ передачи параметров может быть различным в различных вызовах одной и той же процедуры (см. ниже).

■ Локализация процедуры. По умолчанию процедура локализована в минимальной из перечисленных ниже конструкций, охватывающих текст процедуры:

— в ближайшем ⟨закрытом предложении⟩ (⟨контексте⟩ контекстной приставки), содержащем описания объектов, использованных внутри процедуры;

— в ближайшем базированном блоке или модуле.

Локализацию процедуры можно задавать явным образом в ⟨генераторе объекта⟩. Процедуру нельзя вызывать после окончания выполнения ⟨закрытого предложения⟩ (после уничтожения модуля), в котором она локализована. Это ограничение не распространяется на процедуры, реализующие конечные реакции на динамические ситуации. ■

Описание процедур-констант используется в качестве сокращенной формы записи для распространенных случаев описания констант процедурного типа.

⟨идентификация процедуры⟩ ::=
⟨идентификатор⟩ {=⟨доопределение процедуры⟩}

■ Описание процедур-констант имеет вид:
процедура *p* = ДП
где: *p* — ⟨идентификатор⟩, а ДП — ⟨доопределение процедуры⟩ типа *тр*. Если *тр* является стандартным типом, то такое описание эквивалентно:

конст *p*#*стпроц* = ДП

причем возможности определения правой части шире, чем в случае обычного описания отдельно размещенной константы (допускается контекстная приставка и процедурная форма доопределения). В правой части допускается рекурсивное использование описываемого идентификатора, например:

процедура *нод* = **функция** (*а*, *в*)
если *в* = 0
то *abs* (*а*)
иначе *нод* (*в*, *а* **остат** *в*)
все

В случае программно определяемого типа *тр* эквивалентом является следующее описание:

конст *р* : *тр* = ДП

которое в соответствии со стандартными правилами допускает рекурсивное определение процедуры.

Если правая часть ⟨идентификации процедуры⟩ опущена, то такое описание является предварительным описанием констант стандартного процедурного типа, причем допускается, чтобы использование идентификатора предшествовало доопределению константы. Это обеспечивает возможность взаимно рекурсивного определения процедур, например:

процедура *а*, *в*;
процедура
а = **проц** (. . .) **начало** . . *в* . . . **конец**,
в = **проц** (. . .) **начало**. . . *а* . . . **конец**.

Предварительное описание и доопределение процедуры-константы должны располагаться в тексте программы в таком же порядке, что и эквивалентные описания константы, т. е. либо в одной и той же цепочке ⟨описаний⟩, либо предварительное описание — в ⟨типе модуля⟩, а доопределение — в ⟨теле модуля⟩.

Вызов процедуры. Правило для ⟨вызова⟩ процедуры имеет следующий вид:

⟨вызов⟩ ::= ⟨первичное⟩ ({⟨список фактических⟩})
⟨фактический⟩ ::=
⟨выражение⟩ | ⟨доопределение объекта⟩
| **прог** ⟨выражение⟩ | **имя** ⟨переменная⟩
| **имя прог** ⟨переменная⟩

Эта конструкция выполняет вызов процедуры, являющейся значением ⟨первичного⟩. Если ⟨первичное⟩ имеет динамический тип, то его значением должна быть процедура стандартного типа, а иначе ⟨первичное⟩ должно быть типа процедуры. При каждом вызове устанавливается соответствие между фактическими и формальными параметрами. Затем в контексте, состоящем из контекста участка программы, где определен ⟨текст процедуры⟩, и совокупности формальных параметров, выполняется ее ⟨закрытое предложение⟩. При необходимости процедура-функция в заключение выдает значение.

Передача параметров. Первая альтернатива правила для ⟨фактического параметра⟩ соответствует случаю, когда параметр передается «значением». Значение фактического параметра вычисляется при вызове процедуры. Это значение рассмат-

ривается как начальное значение соответствующего формального параметра.

В случае процедур программно определяемого типа контроль параметров производится статически, причем для передачи «значением» формальный параметр должен являться константой. Соотношение типов формального и фактического параметров такое же, как в случае левой и правой частей описания. Выполнение передачи «значением» производится следующим образом. Выполняются описания констант, сформированные из параметров: левой частью описания является соответствующее ⟨описание параметра⟩ из ⟨типа процедуры⟩, а правой частью — соответствующий ⟨фактический⟩. Эти описания вводят идентификаторы, включаемые в контекст, в котором выполняется тело процедуры.

■ В случае процедур стандартного типа число и формат значений фактических параметров должны совпадать с числом и форматом соответствующих формальных параметров (если не предпринято специальных мер по организации процедуры с переменным числом параметров, см. выше). В противном случае в процессе выполнения процедуры может возникнуть ситуация *ошибкаимени*. Для формального параметра формата ф64 допускается, чтобы значение фактического параметра имело формат ф32. В случае формального параметра формата ф32 необходимо, чтобы фактическому параметру предшествовало служебное слово **пак32**.

В случае процедуры с переменным числом параметров указанное ограничение распространяется на постоянную часть параметров. В связи с тем, что значения параметров передаются без какого либо преобразования их формата, может оказаться, что область памяти, содержащая переменную часть, заполнена неоднородно. Это необходимо учитывать при выборе способа индексации вектора переменной части.

Существует три дополнительных способа передачи параметров:

1. **Значение подпрограммой (прог)**. Значение фактического параметра вычисляется при каждом обращении к соответствующему формальному параметру в процессе выполнения процедуры. Вычисление происходит в контексте точки вызова. Формальный параметр может использоваться только в качестве ⟨первичного⟩ (но не переменной).

2. **Именем (имя)**. При входе в процедуру переменная, переданная в качестве фактического параметра, ставится в соответствие формальному параметру. Последний может использоваться внутри процедуры как ⟨переменная⟩ и как ⟨первичное⟩. Каждое обращение (для выборки или для присваивания) к формальному параметру в процессе выполнения процедуры означает обращение к соответствующей фактической переменной.

3. Имя подпрограммой **(имя прог)**. Отличается от способа «именем» тем, что фактический параметр выполняется не один раз при входе в процедуру, а при каждом обращении к соответствующему формальному параметру в процессе работы процедуры. Выполнение происходит в контексте точки вызова.

В случае процедур стандартного типа в каждом из этих трех способов требуется, чтобы формат формального параметра был словным (ф64). При этом допускается любой формат фактического параметра.

В случае процедур программно определяемого типа ⟨описание параметра⟩ задает возможные способы передачи фактического параметра:

— если ⟨описание параметра⟩ имеет вид ⟨описания константы⟩, то соответствующий фактический параметр передается только «значением». Если ⟨описанию константы⟩ предшествует служебное слово **прог**, то допускается также передача параметра способом «значение подпрограммой»;

— если ⟨описание параметра⟩ имеет вид ⟨описания переменной⟩, то соответствующий фактический параметр должен быть переменной и передаваться «именем». Если ⟨описанию переменной⟩ предшествует служебное слово **прог**, то допускается передача способом «имя подпрограммой». ■

Процедурная форма доопределения. Описываемые ниже средства языка позволяют задать доопределение объекта с помощью процедуры, которая выполняется в момент создания объекта.

Доопределение модуля. Различаются два случая доопределения модуля: случай встроенной константы и случай динамической генерации объекта.

Доопределение встроенной константы типа модуля, имеющее вид

(e, $e1$, . . ., ek)

где: e — ⟨выражение⟩, выдающее процедуру доопределения, $e1$, . . ., ek — ⟨выражения⟩ типа $m1$, . . ., mk, выполняется следующим образом:

— выполняется вызов процедуры

e#**функция** (**конст**#$m1$, . . .,#mk; **конст**)#m ($e1$, . . ., ek, *вж*)

где: m — тип доопределяемой константы, *вж* — неявно подставляемый параметр типа *ствж*, локализованный в блоке или модуле, где описана доопределяемая константа;

— элементы модуля, выдаваемого процедурой, становятся элементами доопределяемого модуля.

В приведенной выше записи, а также далее в этом разделе подразумевается, что если какой-либо из параметров процедуры доопр-

ределения имеет динамический тип, то его описание заменено на **конст.**

Если какой-либо из параметров имеет тип модуля, то синтаксический вид соответствующего ⟨выражения⟩ ограничен конструкциями ⟨идентификатор⟩ или ⟨элемент модуля⟩, обозначающими константу, или ⟨внешнее имя⟩, приводящее к программе-объекту типа модуля. Вид ⟨выражения⟩ *е* также ограничен этими конструкциями, которые в данном случае должны обозначать константу статического класса, значением которой является процедура доопределения, или программу-объект, являющуюся процедурой доопределения.

■ З а м е ч а н и е. Если процедура доопределения является внешним объектом, то эти ограничения обеспечивают возможность, не изменяя текста программы, задавать разные режимы трансляции вызова процедуры доопределения (либо путем динамического вызова, либо с помощью комплексации с вызывающей программой), а также позволяют произвести статический контроль типа процедуры доопределения. ■

Г е н е р а т о р о б ъ е к т а типа модуля, имеющий вид:

ген *т* [*локал* : *вж*] **иниц** (*е*, *е*1, . . ., *ek*)

где: *е* — ⟨выражение⟩, задающее процедуру доопределения, *ej* — ⟨выражения⟩ типа *mj* (*j*=1, . . ., *k*), *вж* — атрибут, задающий время жизни, значение которого выбирается по умолчанию или явным образом, выполняется следующим образом:

— выполняется вызов процедуры

е#**функция** (**конст**#*т*1, . . ., #*mk*; **конст**)#*т* (*е*1, . . ., *ek*, *вж*),

— результатом генератора является результат, выдаваемый этим вызовом процедуры.

Р е а л и з а ц и я м о д у л я. Для сокращения записи процедур доопределения модулей введена специальная конструкция ⟨реализация модуля⟩.

⟨реализация модуля⟩ ::=
  {⟨формальный контекст⟩}
  реал ⟨имя типа⟩
  {(⟨список идентификации констант⟩)} ⟨тело модуля⟩
⟨формальный контекст⟩ ::=
  контекст (⟨список идентификации констант⟩)

⟨Идентификация константы⟩ в данном случае должна иметь вид предварительного описания отдельно размещенной константы, причем в ⟨формальном контексте⟩ константа должна иметь тип модуля.

⟨Реализация модуля⟩, имеющая вид:

контекст (*xm*#*l*1, . . ., *xm*#*ml*)
реал *m* (*y*1#*my*1, . . ., *yk*#*myk*)
контекст СЧ **начало** ИЧ **конец**

где: СЧ — описания скрытой части, ИЧ — доопределения элементов интерфейсной части, эквивалентна ⟨тексту процедуры⟩:

функция (**конст** *xx*1#*m*1, . . ., *xxl*#*ml*,
*yy*1#*my*1, . . ., *yyk*#*mlk*; **конст** *вж*)#*m*
(**ген** *m* [*локал* : *вж*] **иниц**
контекст
конст *x*1#*m*1 = *xx*1; *x*1; . . .;
конст *xl*#*ml* = *xxl*;*xl*
конст *y*1#*my*1 = *yy*1, . . .,
yk#*myk* = *yyk*
;СЧ
начало ИЧ **конец**)

Идентификатор *вж* выбирается так, чтобы он отличался от всех идентификаторов, используемых в программе.

Процедурное доопределение процедуры. Различаются два случая доопределения процедуры: случай встроенной константы и случай динамической генерации объекта.

Доопределение встроенной константы типа процедуры, имеющее вид:

(*e*, *e*1, . . ., *ek*)

где: *e* — ⟨выражение⟩, выдающее процедуру доопределения, *ej* — ⟨выражения⟩ модульного типа *mj* (*j*=1, . . ., *k*), выполняется следующим образом:

— выполняется ⟨вызов⟩ процедуры

e#**функция** (**конст**#*m*1, . . .,#*mk*; **конст**)#*m* (*e*1, . . ., *ek*, *вж*)

где: *m* — тип константы, *вж* — имеет такой же смысл, как и в случае доопределения константы типа модуля;

— тело процедурного объекта, выдаваемого процедурой, становится телом доопределяемого объекта.

Синтаксический вид ⟨выражений⟩ *e*, *e*1, . . ., *ek* ограничен, как в случае доопределения модуля.

Генератор объекта типа процедуры:

ген *m* (*локал* : *вж*) **иниц** (*e*, *e*1, . . ., *ek*)

где: *e*, *e*1, . . ., *ek*, *вж* — обозначения, имеющие такой же смысл, как и в случае генератора объекта типа процедуры, выполняется следующим образом:

— выполняется ⟨вызов⟩ процедуры, имеющий такой же вид, как описано выше;

— результат, выдаваемый процедурой, становится значением генератора.

Реализация процедуры. Для сокращенной записи процедур доопределения процедур введена специальная конструкция ⟨реализация процедуры⟩:

⟨реализация процедуры⟩ ::=
  {⟨формальный контекст⟩}
  реал {⟨контекст⟩} ⟨текст процедуры⟩

⟨Реализация процедуры⟩, имеющая вид:

контекст (*x*1##*m*1, . . ., *xk*##*mk*)
реал контекст СЧ ТП

где: СЧ — описания скрытой части, ТП — ⟨текст процедуры⟩ типа *m*, эквивалента ⟨тексту процедуры⟩:

функция (конст *xx*1##*m*1, . . ., *xxk*##*mk*; **конст** *вж*)##*m*
  (**ген** *m* [*локал* : *вж*] **иниц**
    контекст
      конст *x*1##*m*1 = *xx*1; *x*1; . . .
      конст *xk*##*mk* = *xxk*; *xk*
      ;СЧ ТП)

11.2. Параллельные процессы и синхронизация. Выполнение процедуры может происходить в синхронном и параллельном режимах. В первом случае вызывающая процедура передает управление вызываемой и сама приостанавливается; после завершения выполнения вызываемой управление возвращается в точку вызова и выполнение вызывающей возобновляется. Такая схема управления реализуется конструкцией ⟨вызов⟩.

В случае параллельного режима вызывающая процедура лишь активизирует выполнение вызываемой, после чего продолжается выполнение предложений, следующих за тем, которое произвело активизацию. Параллельно с этим происходит выполнение вызванной процедуры.

Действие по активизации выполнения в параллельном режиме называется созданием параллельного процесса. Процедура, вызванная в таком режиме, называется головной процедурой процесса. Головная процедура может в свою очередь вызывать другие процедуры в синхронном или параллельном режиме.

Процесс завершается при завершении его головной процедуры. Кроме нормального завершения, возможно прекращение процесса. Оно происходит тогда, когда выполнение головной процедуры прекращается из-за распространения ситуации.

Подчиненные и независимые процессы. Различаются два основных класса процессов: независимые и подчиненные.

В первом случае не существует какой-либо неявной синхронизации между независимым процессом и процессом, его породившим. В частности, если нет явно заданной синхронизации, то окончание (нормальное завершение или прекращение) независимого процесса не связано с какими-либо моментами выполнения создавшего процесса. Если прекращение произошло вследствие распространения ситуации, то перед ликвидацией процесса выполняется конечная реакция, связанная с данной ситуацией (см. п. 10.2). Если независимый процесс прекратился из-за перехода на глобальную метку, то выдается сообщение об ошибке.

В случае подчиненного процесса (*B*) существует критический блок в создавшем процессе (*A*), охватывающий место запуска *B*. Выполнение критического блока заканчивается не раньше, чем заканчивается выполнение *B*. Если выполнение всех предложений критического блока нормально завершилось до того, как закончился процесс *B*, то процесс *A* ожидает окончания *B*. Выполнение критического блока заканчивается лишь после того, как закончилось выполнение *B*. Поэтому подчиненный процесс может использовать переменные, константы и объекты, локализованные в критическом блоке (и блоках, охватывающих критический), не предпринимая при этом специальных мер по синхронизации.

При нормальном завершении процесса *B* критический блок продолжает выполняться в том случае, если еще остались невыполненные предложения или неоконченные подчиненные процессы.

Если выполнение критического блока прекращается из-за распространения ситуации, то одновременно прекращается выполнение процесса *B* (и других подчиненных процессов этого блока); затем ситуация продолжает распространяться за пределами критического блока в процессе *A*.

Если выполнение процесса *B* прекращается из-за распространения ситуации, то дальнейшее распространение данной ситуации продолжается уже в процессе *A*. В последующем может произойти одно из двух: либо распространение ситуации будет прекращено в процессе *A*, либо процесс *A* будет прекращен. Если *A* в свою очередь подчинен некоторому процессу *C*, то распространение ситуации продолжится в *C*.

В случае, если независимый процесс *A* использует переменные, константы и объекты, локализованные в блоках другого процесса, необходимо учитывать следующее обстоятельство. Окончание выполнения этих блоков и ликвидация соответствующих данных может произойти еще до окончания процесса *A*. Последующее использование в *A* уничтоженных данных приведет к ошибке.

Этого можно избежать, устроив соответствующим образом синхронизацию процессов.

Процесс создается с помощью стандартной процедуры *создпроцесс*, имеющей следующие параметры:

создпроцесс (*класс*, *p*, *x*1, . . ., *xn*)

где *класс* — параметр, задающий класс процесса (0 — независимый, 1 — подчиненный); значением второго параметра является головная процедура процесса; *x*1, . . ., *xn* — это параметры, передаваемые в головную процедуру. Параметры могут передаваться значением или именем *) (см. п. 11.1). Критическим блоком для подчиненного процесса является ближайший базированный блок, охватывающий место его создания.

Синхронизация процессов. Синхронизация осуществляется с помощью объекта типа семафор. Семафор может находиться в состоянии «открыт» или в состоянии «закрыт». Процесс может быть приостановлен на закрытом семафоре. Впоследствии процесс может быть возобновлен.

Существуют две «заготовки» для семафоров. Один семафор находится в открытом состоянии, а другой — в закрытом. Первый семафор является значением стандартной константы *семоткр*, а второй — значением стандартной константы *семзакр*. Для работы с семафором надо одну из заготовок присвоить переменной, и затем эту переменную передавать по имени в операции над семафором. При этом состояние семафора, являющегося значением переменной, будет соответствующим образом меняться. Ниже приводятся операции над семафорами. Идентификатором *с* обозначен параметр, имеющий синтаксическую форму ⟨переменная⟩. Параметр может быть передан не только «именем», но и «значением». Во втором случае следует передавать одноэлементный вектор, содержащий семафор.

закрытьсем (имя *с*). Если семафор открыт, то операция переводит его в закрытое состояние, и процесс продолжается. В противном случае процесс приостанавливается на данном семафоре.

ждать (имя *с*). Если семафор открыт, то процесс продолжается, не закрывая его. В противном случае процесс приостанавливается на данном семафоре.

открытьсем (имя *с*). Если семафор открыт, то его состояние не изменяется. В противном случае семафор открывается и возобновляются все процессы, приостановленные на данном семафоре.

---

*) При передаче по имени, а также при передаче составных объектов (оперативных или внешних) надо учитывать, что выполнение блока, в котором локализован передаваемый объект или переменная, может закончиться раньше, чем выполнение процесса.

Если процессы были приостановлены в результате выполнения ими операции *закрытьсем*, то каждый из них прежде всего повторяет эту операцию.

Независимо от исходного состояния семафора процесс, выполнивший операцию *открытьсем*, продолжается.

пропустить (**имя** *с*). Отличие этой операции от предыдущей состоит в том, что: (а) семафор не открывается, и (б) каждый из возобновленных процессов, не повторяя операцию закрытия, начинает выполнять следующие за ней действия.

11.3. Задача. Особенности задачи:

— задача выполняется в собственной математической памяти;

— задача является той единицей вычислений, на которую начисляется расход ресурсов (объем используемой оперативной и вторичной памяти, время решения, время обмена и пр.);

— задача имеет ряд атрибутов, характеризующих текущий расход и лимит ресурсов, имя задачи и пр. (см. атрибуты задачи).

Конструкция ⟨запуск задачи⟩ используется в двух целях:

— для запуска новой задачи в процессе выполнения процедуры (т. е. для запуска одной задачи из другой);

— для оформления пакетного задания (см. § 15 гл. 3).

⟨запуск задачи⟩ ::=
задача (⟨список установок атрибутов⟩)
⟨текст программы⟩ {(⟨список фактических⟩)}

В результате выполнения ⟨запуска задачи⟩ создается новая задача. При необходимости, используя ⟨список установок атрибутов⟩, можно задать значения атрибутов этой задачи.

В рамках задачи создается независимый процесс, в котором роль головной процедуры играет заданная программа. Этот процесс может в свою очередь создавать процессы и задачи. ⟨Список фактических⟩ содержит параметры, передаваемые в головную процедуру процесса.

Задача ликвидируется, если (а) закончены все созданные в ней независимые процессы, и (б) не осталось прикрепленных к ней ацд- или пм-файлов, для которых задана процедура реакции на активность со стороны терминала (см. п. 8.7 гл. 3). ■

12. Программа

В этом параграфе вводится конструкция, определяющая общую синтаксическую структуру текста программы.

В простейшем случае текст программы — это ⟨текст процедуры⟩ или ⟨закрытый оператор⟩, причем последний рассматривается как сокращенная запись процедуры без параметров:

проц( ) ⟨закрытый оператор⟩

Программа может быть вызвана в диалоговом режиме с терминала пользователя или может быть вызвана из другой программы. В частности, вызывающей программой может являться программа пакетного задания или «управляющая программа», которая организует необходимую последовательность выполнения программ-процедур и/или создает объединение взаимозависимых модулей. Ниже разобраны примеры программирования вызова программ из пакетных заданий. Для более подробного знакомства с возможностями языка в части обработки программ см. в гл. 3: § 1, пп. 2.1, 5.4 и § 12—15. Примеры объединения модулей приведены в § 20.

Программа задания на языке Эль-76 может быть представлена в виде обычной программы. Ниже приведен пример программы задания. В примере использовано то свойство, что текст одной программы может содержать текст другой программы. Последний обозначается с помощью конструкции ⟨изображение текстового файла⟩ (см. гл. 3 § 13).

Пример. Программа задания:

```
начало
  . . .
  тфайл (имяяз: "авт")
  *!!*
  текст
  проблемной
  программы
  !!
  . . .
конец
```

Здесь в тексте программы задания содержится ⟨изображение текстового файла⟩, начинающееся служебным словом **тфайл**. Между ограничителями *!!* и !! заключен текст вложенной программы. Значением данной конструкции является указатель (внешнего объекта), приводящий к изображенному текстовому файлу.

Вызов программы, как и вызов процедуры, осуществляется с помощью конструкции ⟨вызов⟩. Один из возможных типов значения ⟨первичного⟩ — это указатель файла текста программы. В результате происходит вызов процедуры, изображенной в тексте программы *). Фактические параметры передаются обычным образом.

*) В этом случае система осуществляет неявную трансляцию текста программы. Поэтому среди атрибутов текстового файла, т. е. в скобках после слова **тфайл**, надо задавать атрибут *имяяз*, определяющий внешнее имя транслятора используемого языка программирования. Отметим, что средствами языка можно явным образом задать такие действия, как вызов транслятора, «запись» результирующей программы в архив и проч.

Пример. Вызов программы из задания:

```
начало
  . . .
  тфайл (имяяз : "авт")
  *!!*
  проц (. . .)
  начало
  . . .
  конец
  !!
  (. . .фактические. . .)
конец
```

Этот же пример можно запрограммировать иначе:

```
начало
  . . .
  конст мояпрог = % описание константы
     тфайл (имяяз  : "авт")
     *!!*
     проц (. . .)
     начало
     . . .
     конец
     !!;
  . . .
  мояпрог (. . .фактические. . .);
  . . .
конец
```

12.1. Программа-объект. Правило для организации текста программы имеет вид:

```
⟨текст программы⟩ ::=
    ⟨описание константы⟩ | ⟨описание типа⟩
  | ⟨закрытый оператор⟩ | ⟨программа-вычисление⟩
⟨программа-вычисление⟩ ::=
  программа {⟨описание⟩;} . . . {⟨оператор⟩} . . . ⟨выражение⟩
  конец
```

Программа — это объект, содержащийся в файле объектного кода. Файлом объектного кода называется файл, имеющий соответствующее значение атрибута *типфайла*. Этот объект образуется в результате трансляции ⟨текста программы⟩.

Существуют две разновидности программ: программа-объект и программа-вычисление. Программа-объект является статическим объектом типа *тип*, типа процедуры или типа модуля. Такая программа задается с помощью ⟨описания типа⟩ или ⟨опи-

сания константы), задающих описание одной константы статического класса (см. п. 7.10). Описание выполняется в результате трансляции ⟨текста программы⟩. Программа-объект является значением полученной константы.

В ⟨описании типа⟩ и ⟨описании константы⟩ процедурного типа можно опускать левую часть описания вместе с символом «=». ⟨Текст программы⟩ в виде ⟨закрытого оператора⟩ представляет собой сокращение записи вида

проц ( ) ⟨закрытый оператор⟩

Раздельная трансляция. Доопределение константы типа процедуры и типа модуля можно задавать в виде отдельно транслируемого ⟨текста программы⟩. В этом случае в основной программе должно быть дано предварительное описание встроенной константы процедурного или модульного типа или процедуры-константы. ⟨Текст программы⟩, доопределяющий эту константу, представляет собой ⟨описание константы⟩, причем в левой части описания вместо ⟨идентификатора⟩ указывается имя элемента программы, имеющее вид

⟨внешнее имя⟩.⟨идентификатор⟩{.⟨идентификатор⟩}...

⟨Внешнее имя⟩ является архивным именем программы-объекта, содержащей предварительное описание, а последующая цепочка ⟨идентификаторов⟩, разделенных символом «.», однозначно селектирует доопределяемую константу по структурной вложенности описания процедур и модулей. Последний ⟨идентификатор⟩ цепочки является идентификатором доопределяемой константы. ⟨Текст программы⟩, доопределяющей константу, транслируется в контексте участка основной программы, где находится предварительное описание этой константы.

Пример. Раздельная трансляция тела модуля. Пусть основная программа имеет вид

```
проц (. . .параметры. . .)
начало
  процедура p = проц (. . .параметры. . .)
    начало
     конст a : тм, % предварительное описание константы
     . . .         % типа модуля тм
    конец;
  . . .
конец.
```

Если эта программа находится в архиве под именем *//мояпрог*, то текст раздельно транслируемого доопределения константы *a* имеет вид

конст *//мояпрог.p.a* = . . . тело модуля . . .

Замечание. В синтаксической структуре имени элемента программы допускаются дополнительные обозначения, идентифицирующие версию основной программы.

12.2. Программа-вычисление. Программа-вычисление состоит из двух частей:

— первая часть — это последовательность описаний констант статического класса. Выполнение этих описаний происходит при трансляции программы и приводит к созданию модуля статического класса;

— вторая часть задает действия, выполняемые при открытии программы.

12.3. Вызов программы. Программу можно открыть и выполнить. Открытие осуществляется либо неявным образом, либо явно (см. гл. 3 п. 2.1). При открытии происходит следующее:

— если открывается программа-объект, то в качестве результата выдается данный объект типа *тип*, процедура или модуль;

— если открывается программа-вычисление, то выполняется последовательность ⟨операторов⟩ программы, и значение заключительного ⟨выражения⟩ выдается в качестве результата открытия.

Значением ⟨первичного⟩ в конструкции ⟨вызов⟩ может быть не только процедура, но и указатель, приводящий к программе, или к файлу объектного кода программы, или к файлу текста программы. В этих случаях действия, связанные с открытием программы (и если необходимо — трансляция текста), производятся неявно. Далее полученная процедура выполняется обычным образом. Например, если значение переменной или константы ϕ приводит к файлу объектного кода или текста некоторой программы, то вызов этой программы с некоторыми параметрами x, y можно записать так: $\phi(x, y)$.

Аналогичным образом программа может быть использована в качестве головной процедуры параллельного процесса (см. п. 11.2).

13. Форматный обмен

В этом параграфе описывается один из существующих в языке способов обмена — форматный обмен. Особенность форматного обмена состоит в том, что он осуществляет: (а) вывод таких объектов, как числа и наборы, с преобразованием их из внутреннего представления в литерное и (б) ввод с обратным преобразованием из литерного представления во внутреннее. Остальные способы осуществляют двоичный обмен, при котором такое преобразование не производится (см. гл. 3 § 7—11).

Форматный обмен осуществляется с текстовым файлом. Последний представляет собой один из возможных вариантов внут-

ренней организации файла. Структура текстового файла описана в гл. 3 § 12. Здесь приведены краткие предварительные сведения.

Текстовый файл представляет собой последовательность строк, состоящих из элементов литерного формата. Различаются файлы строк фиксированной длины и файлы строк плавающей длины. В первом случае длина всех строк файла одинакова и равна значению атрибута *длинстрок*. Во втором случае длина строки может изменяться. Все строки файла могут быть пронумерованы или нет — в зависимости от значения атрибута *длинфнс*. Каждая строка нумерованного файла содержит собственный фиксированный номер строки.

Текущая позиция в файле — это позиция текущей литеры в текущей строке. Для отслеживания текущей позиции используется оперативный объект связи с файлом, имеющий тип «позиционная переменная». Для случая форматного обмена существенны следующие атрибуты этого объекта: *фнс* — фиксированный номер текущей строки, *шагфнс* — шаг нумерации. Эти атрибуты определены только для нумерованных файлов.

Перед началом форматного обмена с некоторым файлом необходимо создать позиционную переменную. Для этого используется конструкция ⟨генератор объекта связи⟩ (см. гл. 3 п. 2.1). Например, пусть значение переменной (константы, формального параметра) *ф* приводит к файлу *). Тогда в результате выполнения оператора

позпф := *позп*(*ф*)

переменной *позпф* присваивается объект, являющийся позиционной переменной, который используется в процессе форматного обмена с файлом.

Другой способ использует то свойство, что параметром генерации позиционной переменной может быть не только ранее созданный файл, но и одна из стандартных констант, обозначающих тип внешнего устройства (ацпу, вывод на перфокарты и т. д.). В этом случае одновременно создается и файл, и позиционная переменная. Например, пусть дана переменная *позпеч*, тогда в результате выполнения оператора

позпеч := *позп* (*ацпу*)

значением переменной *позпеч* станет объект, который является позиционной переменной вновь созданного файла на алфавитно-цифровом печатающем устройстве.

Первоначально текущая позиция установлена на начало файла. Используя операции позиционирования (см. гл. 3 п. 10.1), ее мож-

*) Здесь подразумевается, что значение *ф* — это оперативный объект, связанный с файлом.

явным образом перемещать на конец файла или возвращать на начало.

Позиционная переменная является первым параметром операций форматного обмена. В процессе обмена происходит перемещение текущей позиции. При выводе последовательность литер, получившаяся в результате преобразования, помещается в текущую строку, начиная с позиции текущего элемента. При вводе последовательность литер, начинающаяся с текущей позиции, преобразуется в значение *) и присваивается заданной переменной. В обоих случаях текущая позиция сдвигается на элемент, следующий за последним обработанным. Существуют средства перехода на следующую строку и средства явной установки позиции текущего элемента строки.

О б м е н с м а с с и в о м. Форматный обмен можно осуществлять не только с файлом, но и с вектором элементов литерного формата. Вектор рассматривается как файл, состоящий из одной строки. Операция обмена обрабатывает некоторое количество элементов и выдает подвектор, элементами которого являются элементы необработанной части исходного вектора. Этот подвектор будем называть остатком исходного вектора.

У п р а в л е н и е о б м е н о м. Существуют две разновидности форматного обмена: обмен, управляемый форматом, и обмен, управляемый данными.

В случае управления форматом преобразование осуществляется с помощью ⟨формата обмена⟩, который явным образом задается в операции. Формат описывает структуру и размер последовательности литер и тем самым задает действия, связанные с преобразованием «значение ↔ последовательность литер».

В случае управления данными обмен осуществляется с помощью одного из стандартных форматов обмена. Формат явно не указывается в операции, но выбирается по умолчанию. При выводе выбор определяется типом и форматом выводимого значения. При вводе выбор зависит от (а) структуры вводимой последовательности литер и (б) формата переменной, которой необходимо присвоить введенное значение.

О ш и б к и. В описании семантики указаны обстоятельства, приводящие к возникновению ошибок. Стандартная реакция системы на ошибки описана в п. 13.5.

13.1. Операции форматного обмена. В языке введены четыре операции форматного обмена: *запф* и *читф* — для обмена с файлом, *запфм* и *читфм* — для обмена с массивом.

---

*) Здесь и далее в этом параграфе под термином «значение» понимается объект, полученный после преобразования при вводе, либо выводимый объект (до преобразования в литерное представление).

⟨форматный обмен⟩ ::=
запф (⟨выражение⟩, ⟨список элементов вывода⟩)
| *читф* (⟨выражение⟩, ⟨список элементов ввода⟩)
| *запфм* (⟨массив обмена⟩, ⟨список элементов вывода⟩)
| *читфм* (⟨массив обмена⟩, ⟨список элементов ввода⟩)
⟨элемент вывода⟩ ::=
⟨выражение⟩ {: ⟨формат обмена⟩} |: ⟨позиционирование⟩
⟨элемент ввода⟩ ::=
имя ⟨переменная⟩ {: ⟨формат обмена⟩}
| ⟨выражение⟩ {: ⟨формат обмена⟩}
| :⟨позиционирование⟩
⟨массив обмена⟩ ::= ⟨выражение⟩ | **имя** ⟨переменная⟩

Значением первого ⟨выражения⟩ операций *запф* и *читф* является позиционная переменная.

Значением первого ⟨выражения⟩ операций *запфм* и *читфм* является вектор. Этот параметр может быть передан «значением» или «именем» (см. п. 11.1). В случае передачи «именем» по окончании выполнения операции значением переданной переменной становится остаток исходного вектора.

⟨Список элементов вывода⟩ и ⟨список элементов ввода⟩ может содержать элементы обмена, управляемого форматом, и/или элементы обмена, управляемого данными. Элементы первой разновидности содержат компоненты «:⟨формат обмена⟩». В элементах второй разновидности такой компоненты нет.

⟨Выражение⟩, входящее в состав ⟨элемента вывода⟩, может задавать выводимое значение или массив выводимых значений. В первом случае значение ⟨выражения⟩ отлично от массива. Во втором случае оно является массивом, отличным от набора. Наборы обрабатываются как одиночное значение. При выводе, управляемом форматом, значения всех элементов массива выводятся под управлением одного и того же формата, причем обработка каждого значения осуществляется по правилам обработки одиночного значения.

⟨Переменная⟩, входящая в состав ⟨элемента ввода⟩, задает переменную, которой присваивается вводимое значение. Присваивание, в том числе контроль типов и преобразование типа значения, выполняется по правилам обычного ⟨присваивания⟩ (см. § 8). Если не соблюдается соотношение типов или невозможно преобразование типа, возникает «ошибка значения». ⟨Выражение⟩ задает массив, элементам которого присваиваются вводимые значения. Как и в случае вывода, обработка каждого элемента массива происходит по правилам обработки одиночной переменной.

Элемент обмена вида «:⟨позиционирование⟩» осуществляет перемещение текущей позиции, в частности, переход на следующую

строку. Такой элемент не производит ввод или вывод какого-либо значения. Здесь, однако, можно в явном виде задать последовательность литер, которая появится в выводимой строке.

Элементы обмена выполняются слева направо в порядке текстуального расположения. ⟨Выражения⟩ и ⟨переменные⟩ выполняются один раз перед началом операции. Элементы n-мерного массива *) обрабатываются в следующем порядке: сначала последовательно обрабатываются элементы с индексами l_1, l_2, . . ., l_{n-1}, i_n, где: l_1, l_2, . . ., l_{n-1} — нижние границы по соответствующим измерениям, а i_n пробегает весь допустимый диапазон значений индекса по n-му измерению, начиная от нижней границы и заканчивая верхней; затем индекс по измерению $n-1$ увеличивается на единицу и процесс повторяется и т. д. Таким образом осуществляется обход всех элементов массива. Если при выводе оказывается, что значением какого-либо элемента массива в свою очередь является указатель массива, то последний массив проходит такой же процесс обработки, а затем продолжается обработка первого. Исключение составляет случай вывода по ⟨трафарету тегированного⟩.

13.2. Обмен, управляемый данными.

Вывод. Обычно вывод начинается с текущей позиции. Если выводимое число не умещается на текущей строке, то предварительно выполняется переход на следующую строку. Если число помещается не в начало строки, то предварительно выводится один разделяющий пробел.

Тип целое. Выводимое число преобразуется в последовательность литер, размер которой в зависимости от формата числа равен: для ф32 — 11 литер (знак + 10 цифр), для ф64 — 20 литер (знак + 19 цифр); начальные нули заменяются на пробелы и знак помещается перед первой значащей цифрой.

Тип вещественное. Выводимое число преобразуется в последовательность литер, имеющую вид TU.V*e*TP, где: T — знак числа или порядка, U — отличная от нуля цифра целой части мантиссы, V — дробная часть мантиссы, *e* — символ порядка, P — порядок. Размер дробной части и порядка зависит от формата числа:

ф32: V—7 цифр, P — 2 цифры;
ф64: V—16 цифр, P — 2 цифры;
ф128: V—33 цифры, P — 5 цифр.

Начальные нули порядка заменяются на пробелы, знак порядка помещается перед первой значащей цифрой.

Тип набор. Если ⟨выражение⟩ имеет динамический тип и значение типа набор, значение преобразуется в целое (см. п. 7.4)

---

*) Если массив получен с помощью ⟨преобразования формы⟩ (см. § 15), то он должен быть прямоугольным параллелепипедом.

и выводится как целое число. Если преобразование приводит к ситуации *неверныйоперанд*, то возникает «ошибка значения». ⟨Выражение⟩ может иметь тип *лог*, *циф* или *лит*. В этом случае значение выводится по правилам вывода элемента массива элементов строчного формата.

Массив. Если массив состоит из элементов простого формата, то значения элементов массива выводятся по правилам вывода чисел и наборов.

В случае, если массив состоит из элементов строчного формата, каждый элемент выводится как последовательность, состоящая из одной литеры. Литера представляет собой ⟨двоичную цифру⟩ для формата ф1, ⟨шестнадцатеричную цифру⟩ для формата ф4 и ⟨литеру⟩ для формата ф8. Разделяющий пробел перед литерой не выводится. При необходимости выполняется переход на следующую строку.

Попытка вывода значений других типов приводит к «ошибке значения».

Ввод. Размер и структура вводимой последовательности литер зависит от того, что собою представляет элемент ввода.

Переменная простого формата. Начиная с текущей позиции, отыскивается первая литера, отличная от пробела (при этом, если нужно, происходит переход на следующую строку). Далее набирается последовательность литер, подчиняющаяся синтаксису ⟨целого⟩ или ⟨вещественного⟩. Числу может предшествовать знак, возможно, отделенный от числа пробелами. Допускаются пробелы после знака порядка и символа порядка *е*. Это число преобразуется во внутреннее представление и присваивается переменной. Если последовательность литер не подчиняется синтаксису ⟨целого⟩ либо ⟨вещественного⟩, возникает «ошибка литеры».

Переменная типа *лог*, *циф* или *лит*. Такая переменная обрабатывается как элемент массива строчного формата. Переменная динамического типа строчного формата, поле переменной и поле вектора (см. гл. 5) обрабатываются следующим образом. Как в случае переменной простого формата, отыскивается целое число без знака, преобразуется во внутреннее представление и присваивается переменной. «Ошибка значения» возникает, если величина числа больше, чем $2^{ф}-1$, где: ф — формат переменной.

Массив. Если массив состоит из элементов простого формата, то каждый его элемент обрабатывается как переменная простого формата. Если массив состоит из элементов строчного формата, каждый его элемент обрабатывается следующим образом:

1. Форматы ф1 и ф4: начиная с текущей позиции, отыскивается первая литера, отличная от пробела (при необходимости выполняется переход на следующую строку). В случае формата ф1 литера

должна быть ⟨двоичной цифрой⟩, а в случае формата ф4 — ⟨шестнадцатеричной цифрой⟩, иначе возникает «ошибка литеры». Литера преобразуется во внутреннее представление, соответствующее двоичным или шестнадцатеричным цифрам, и присваивается элементу массива.

2. Формат ф8: литера, находящаяся в текущей позиции, присваивается элементу массива. При этом, если в текущей строке нет больше литер, то предварительно выполняется переход на следующую строку.

13.3. Позиционирование. ⟨Позиционирование⟩ обеспечивает следующие основные возможности: установку новой текущей позиции, вывод заданной последовательности литер.

⟨позиционирование⟩ ::= ⟨литерал⟩{⟨размещение⟩}...
| ⟨размещение⟩...
⟨размещение⟩ ::= {⟨повторитель⟩} ⟨код размещения⟩{⟨литерал⟩}
⟨литерал⟩ ::=
{⟨повторитель⟩} "⟨литера⟩..." {⟨повторитель⟩ "⟨литера⟩..."}...
⟨код размещения⟩ ::= *x* | *y* | *k* | *l* | *p*
⟨повторитель⟩ ::= ⟨целое⟩ | *n* (⟨выражение⟩)

В последнем правиле буква *n* — это латинское N.

Установка позиции задается с помощью компоненты ⟨размещение⟩. Конкретные действия обозначаются соответствующим ⟨кодом размещений⟩:

x — текущая позиция смещается вправо на один элемент. Если при выводе это приводит к появлению незаполненного элемента, то в него помещается пробел.

y — текущая позиция смещается влево на один элемент.

k — текущим элементом становится элемент строки с номером, равным значению ⟨целого⟩ или ⟨выражения⟩ из ⟨повторителя⟩. Если при выводе это приводит к появлению незаполненных элементов, то в них помещаются пробелы.

l — текущая позиция перемещается в начало следующей строки. При выводе имеют место следующие особенности. В случае файла строк фиксированной длины незавершенная часть выводимой строки заполняется пробелами. В случае файла строк плавающей длины выводится последовательность литер, заключенная между началом строки и текущей позицией. При выводе в нумерованный файл новой строке присваивается номер, равный сумме текущих значений атрибутов *фнс* и *шагфнс*. Это же число присваивается атрибуту *фнс*. В операциях обмена с массивом *запфм* и *читфм* подобный ⟨код размещения⟩ игнорируется.

p — происходит переход на первую строку следующей страницы. Такое действие имеет смысл только для ацпу-файлов. В остальных случаях подобный ⟨код размещения⟩ игнорируется.

Компонента ⟨литерал⟩ имеет следующий смысл. При выводе последовательность литер, заключенная между кавычками, выводится, начиная с текущей позиции, как массив литер (см. п. 13.2). При вводе проверяется, совпадает ли последовательность литер, начинающаяся с текущей позиции, с последовательностью, заключенной между кавычками (в процессе сравнения, если надо, осуществляется переход на следующую строку). В случае несовпадения возникает «ошибка литеры».

Число повторений. ⟨Целое⟩ или значение ⟨выражения⟩ в ⟨повторителе⟩ задает число повторений соответствующего действия (операции позиционирования, операции вывода или сравнения ⟨литерала⟩). Если ⟨повторитель⟩ опущен, действие выполняется один раз. Исключение составляет случай ⟨кода размещения⟩ *k*, когда ⟨повторитель⟩ имеет другой смысл (см. выше) и должен присутствовать обязательно.

13.4. Обмен, управляемый форматом.

Вывод. Значение преобразуется в последовательность литер, структура которой определяется ⟨форматом обмена⟩. Каждый формат накладывает определенные ограничения на тип и величину значений, выводимых по данному формату. Если эти ограничения не выполняются, возникает «ошибка значения». Последовательность литер выводится, начиная с текущей позиции.

Ввод. Последовательность литер, начинающаяся с текущей позиции, проходит обратное преобразование. Структура вводимой последовательности должна соответствовать ⟨формату обмена⟩, иначе возникает «ошибка литеры». Значение, полученное в результате преобразования, присваивается заданной переменной.

Ввод/вывод массива. Если особо не оговорено, то при выводе массива значение каждого элемента массива выводится по правилам обработки одиночного значения. Аналогично при вводе каждый элемент массива обрабатывается как одиночная переменная. Обработка всех элементов происходит под управлением одного и того же ⟨формата обмена⟩.

⟨формат обмена⟩ ::= {⟨вставка⟩} ⟨трафарет⟩
⟨трафарет⟩ ::=
⟨трафарет числа⟩ | ⟨трафарет целого⟩
| ⟨трафарет вещественного⟩ | ⟨трафарет литерного⟩
| ⟨трафарет логического⟩ | ⟨трафарет двоичного⟩
| ⟨трафарет тегированного⟩

Каждый конкретный ⟨трафарет⟩ управляет вводом/выводом последовательности литер, имеющей некоторую специфическую форму. Например, ⟨трафарет вещественного⟩ описывает последовательность, имеющую форму вещественного числа с плавающей точкой (экспоненциальная форма) или числа с фиксированной точкой.

По ⟨трафаретам⟩, связанным с вводом/выводом чисел, можно определить общее количество цифр в числе. Это количество ограничено; для целых — не более 19 цифр, а для вещественных — не более 34 цифр мантиссы и не более 5 цифр порядка.

В с т а в к а. При выводе ⟨вставка⟩ предписывает поместить в выводимую последовательность заданные литеры. При вводе проверяется, что заданные литеры содержатся во вводимой последовательности. Далее эти литеры из рассмотрения исключаются и не принимают участие в формировании внутреннего представления вводимого значения. Если результат проверки отрицательный, то возникает «ошибка литеры».

⟨вставка⟩ ::= ⟨литерал⟩ {⟨смещение⟩}... | ⟨смещение⟩...
⟨смещение⟩ ::= {⟨повторитель⟩} *x* {⟨литерал⟩}

Компонента ⟨смещение⟩ имеет такой же смысл, как ⟨размещение⟩ с ⟨кодом размещения⟩ *x*. ⟨Повторитель⟩ используется не только во ⟨вставках⟩, но и с тем же смыслом — внутри ⟨трафаретов⟩. Исключение составляет ⟨трафарет числа⟩, где ⟨повторитель⟩ используется в ином смысле (см. ниже).

■ Значения ⟨выражений⟩ всех ⟨повторителей⟩ всех элементов обмена выполняются один раз перед началом обмена. На ⟨повторители⟩ ⟨литералов⟩, входящих в состав ⟨формата обмена⟩, наложено следующее ограничение: здесь допускаются только ⟨выражения⟩ статического класса. Это ограничение не распространяется на ⟨литералы⟩, входящие в состав ⟨позиционирования⟩. ■

Трафарет числа. Трафарет числа управляет вводом/выводом последовательности литер в форме целого числа, числа с фиксированной точкой или числа с плавающей точкой. Он обеспечивает наиболее простой способ ввода/вывода чисел, являющийся промежуточным между обменом, управляемым данными, и обменом, управляемым форматом.

⟨трафарет числа⟩ ::=
{⟨повторитель⟩}*g*{(⟨выражение⟩{,⟨выражение⟩
{,⟨выражение⟩}})} {⟨вставка⟩}

В ы в о д. Форма выводимой последовательности определяется количеством ⟨выражений⟩ в скобках: 1 — целое число, 2 — число с фиксированной точкой, 3 — число с плавающей точкой. Если скобки и заключенные в них ⟨выражения⟩ опущены, то вывод производится по правилам обмена, управляемого данными. Обозначим значения ⟨выражений⟩: *ww*, *wf*, *we*.

Ц е л о е ч и с л о. *ww* задает общий размер последовательности. Незначащие нули числа заменяются на пробелы. Знак помещается перед первой значащей цифрой. Отрицательный знак *ww* означает, что знак включается в общий размер и выводится только

для отрицательных чисел. Если *ww* равно нулю, то выводится последовательность минимальной возможной длины (знак выводится только для отрицательных чисел). Если *ww* отлично от нуля и выводимое число не умещается в размер *ww*, то возникает «ошибка значения».

Пример.

По трафарету *g*(—4) можно вывести числа *):

␣␣␣0 ␣␣12 ␣—12 1234

по трафарету *g*(4) можно вывести

␣␣+0 ␣+12 ␣—12

но число 1234 вывести нельзя.

По трафарету *g*(0) можно вывести

0 12 123 1234

Число с фиксированной точкой. *ww* задает общий размер последовательности, *wf* — число цифр после десятичной точки. Знак *ww* и значение *ww*, равное нулю, трактуются так же, как и в предыдущем случае. Если величина числа такова, что перед десятичной точкой не умещаются все цифры целой части, то количество позиций, отводимых для дробной части, сокращается. Если это количество обратилось в нуль, но число тем не менее не умещается, возникает «ошибка значения».

Пример.

По трафарету *g*(—6,3) можно вывести

␣1.234 —1.234 12.345 123.45
1234.5 ␣12345 123456

В последних трех случаях число позиций после точки сократилось.

По трафарету *g*(0, 3) можно вывести

1.234 —1.234 123.456

Число с плавающей точкой. *ww* задает общий размер последовательности, *wf* — число цифр дробной части, *we* — число цифр порядка. Последовательность литер имеет форму TU.V*e* TP, где T — знак мантиссы и порядка, U — поле цифр целой части, V — поле цифр дробной части, *e* — символ порядка, P — поле цифр порядка.

Независимо от знака *ww* под знак мантиссы всегда отводится первая позиция. Если знак *ww* — отрицательный, то положительный знак мантиссы кодируется пробелом. Мантисса выравнивается относительно десятичной точки таким образом, чтобы следу-

*) Символ ␣ означает пробел в выводимой последовательности.

ющая за знаком первая цифра целой части была ненулевая. Далее выводятся остальные цифры целой и дробной частей мантиссы. Порядок выводится по правилам вывода целого числа по трафарету *g*(*we*). Знак *we* трактуется так же, как описано выше. Если *we* равно нулю, то предпринимается попытка представить по возможности большее число значащих цифр числа. С этой целью размеры поля порядка и поля целой части выбираются так, чтобы первое было минимально необходимым, а второе—максимально возможным.

Если величина порядка такова, что он не умещается в поле порядка, число позиций, отводимых для мантиссы, сокращается; сначала сокращается число позиций поля дробной части, а затем — число позиций поля целой части. При этом, однако, требуется, чтобы осталось по крайней мере одна цифра целой части, иначе возникает «ошибка значения».

Размер выводимой последовательности всегда равен *ww*, которое в рассматриваемом случае должно быть отлично от 0.

Пример.

По трафарету *g* (10, 3, 3) можно вывести

—1.234*e*␣+0 +1.234*e*—12 +1.23*e*+456 +1*e*␣+12345

В двух последних случаях число позиций поля дробной части сократилось.

По трафарету *g* (6, 1, 0) можно вывести

—123*e*4 +12*e*34 +11*e*—3

В последнем случае подразумевается, что выводимое число равно, например, 0.011.

По ⟨трафарету числа⟩ можно выводить целые и вещественные числа. Вывод вещественного числа по трафарету *g* (*ww*) эквивалентен выводу по трафарету *g* (*ww*, 0).

Массив. Вывод массива производится поэлементно под управлением одного и того же трафарета.

⟨Повторитель⟩ позволяет управлять количеством чисел, располагающихся на одной строке файла. Пусть значение ⟨повторителя⟩ равно *к*. Тогда в строке, начиная с левой крайней позиции, равномерно размещается *к* чисел; каждое число занимает *ww* позиций. *ww* в данном случае должно быть отлично от нуля. Каждая строка массива выводится с новой строки файла. Под строкой массива здесь понимается совокупность элементов *n*-мерного массива, получаемая путем фиксации первых *n*—1 индексов.

Перед выводом начальной строки массива выводится ⟨вставка⟩, если она есть; затем проверяется, находится ли текущая позиция в начале строки. Если нет — осуществляется переход на новую строку. После вывода последней строки всегда происходит переход на новую строку.

Если выводится не массив, а одиночное значение, то ⟨повторитель⟩ игнорируется.

В в о д. При вводе по ⟨трафарету числа⟩ все компоненты трафарета, кроме ⟨вставки⟩, игнорируются, и ввод происходит по правилам обмена, управляемого данными. ⟨Вставка⟩, если она есть, обрабатывается обычным образом.

■ *Трафарет целого.* ⟨Трафарет целого⟩ управляет вводом/выводом последовательности литер, имеющей форму целого числа со знаком или без знака.

```
⟨трафарет целого⟩ ::= {⟨поле знака⟩} ⟨поле цифр⟩
⟨поле знака⟩ ::=
   ⟨знак⟩ {⟨вставка⟩}|⟨поле плавающего знака⟩{⟨вставка⟩}
⟨поле цифр⟩ ::= ⟨подполе цифр⟩...
⟨подполе цифр⟩ ::=
     {⟨повторитель⟩} {s} d {⟨вставка⟩}
   | {⟨повторитель⟩} z {⟨условная вставка⟩}
   | {⟨повторитель⟩} z ⟨вставка⟩
⟨поле плавающего знака⟩ ::=
   ⟨подполе плавающего знака⟩...⟨знак⟩
⟨подполе плавающего знака⟩ ::=
     {⟨повторитель⟩} f {⟨условная вставка⟩}
   | {⟨повторитель⟩} f ⟨вставка⟩
⟨условная вставка⟩ ::= c ⟨вставка⟩
```

В трафарете возможны следующие редакционные особенности:

1. Замена незначащих и значащих нулей на пробелы (с помощью компоненты ⟨подполе цифр⟩ с *z*-спецификацией).

2. Замена незначащих нулей на пробелы и вставка знака перед первой значащей цифрой (с помощью компоненты ⟨поле плавающего знака⟩).

3. Редакционные вставки (с помощью компоненты ⟨вставка⟩ и ⟨условная вставка⟩).

4. Отбрасывание некоторых цифр числа (с помощью компоненты ⟨подполе цифр⟩ с *s*-спецификацией).

Для целого без знака (в трафарете опущено ⟨поле знака⟩) в случае, если в трафарете не заданы редакционные вставки и отбрасывание цифр, соответствующая последовательность литер представляет собой последовательность цифровых позиций (поле цифр)

yy. . .y

где *y* — пробел или цифра.

В простейшем случае ⟨трафарет целого⟩ состоит из одного ⟨подполя цифр⟩ с *d*-спецификацией или одного ⟨подполя цифр⟩ с *z*-спецификацией. Размер вводимого или выводимого поля цифр задается ⟨повторителем⟩.

d-спецификация задает безусловное подполе цифр. Все позиции безусловного подполя заполнены цифрами числа. Незначащие нули, если таковые есть, представлены символом 0. Например, по трафарету 5*d* число 2 представляется в форме 00002.

z-спецификация задает условное подполе цифр. При выводе начальные нули условного подполя (незначащая часть подполя) заменяются на пробелы. Во вводимой строке начальные нули могут быть представлены нулями или пробелами. Значащая часть подполя, т. е. часть, начинающаяся с первой значащей цифры, заполнена цифрами числа. Например, по трафарету 5*z* число 2 представляется в форме *) ␣␣␣␣2.

Поле цифр в общем случае состоит из безусловных и условных подполей. Размер поля цифр равен сумме размеров составляющих его подполей (*wd*+*wz*, где *wd* — суммарный размер безусловных подполей, *wz* — суммарный размер условных подполей). Позиции подполей заполняются по указанным выше правилам. В частности, начальные нули условного подполя при выводе заменяются на пробелы независимо от того, является ли оно начальным подполем поля цифр или каким-либо внутренним подполем. Следует учитывать, что последовательность условных подполей (возможно, включающая редакционные вставки) образует единое условное подполе.

Например, по трафарету 3*z*2*d* число 21012 представляется в форме 21012, а число 1012 — формой ␣1012, число 12 — формой ␣␣␣12, число 2 — формой ␣␣␣02, число 0 — формой ␣␣␣00. Трафареты 2*zz*2*d* и *zzz*2*d* эквивалентны 3*z*2*d*.

Вставки. Различаются безусловные и условные редакционные вставки. Безусловная редакционная вставка задается компонентой ⟨вставка⟩ (семантика ⟨вставки⟩ описана выше). Например, по трафарету "*x*="4*zd* число 12 представляется формой *x*= =␣␣␣12, по трафарету *zd*"—"*zd*"—"3*zd* число 6051980 представляется формой ␣6—␣5—1980.

В условном подполе цифр может быть задана условная редакционная вставка. Литеры вставки помещаются в соответствующие позиции подполя только в том случае, если вставка попадает в значащую часть подполя. Если же вставка попадает в незначащую часть подполя, соответствующие ей позиции заполнены пробелами. Так, с помощью трафарета 3*zc*", "3*zc*", "2*zd* можно разделить запятой разряды, соответствующие миллионам, тысячам и сотням.

Пример.

1. Число 100000000 представляется формой 100,000,000.
2. Число 1000000 представляется формой ␣␣1,000,000.

---

*) Здесь и далее символ ␣ означает пробел в выводимой или вводимой последовательности литер.

3. Число 1000 представляется формой ␣␣␣␣␣␣1,000 (первая запятая заменена на пробел).

При вводе требуется, чтобы в позициях, соответствующих условной вставке, находились литеры вставки или пробелы.

Отбрасывание цифр. С помощью *s*-спецификации можно задать отбрасывание всех цифр какого-либо безусловного подполя цифр. Это означает, что при выводе цифры соответствующего подполя числа отбрасываются и не появляются в выводимой последовательности. При вводе данное подполе вставляется во вводимую последовательность. Позиции вставляемого подполя заполнены нулями. Например, с помощью трафарета *2zd6sd* можно задать масштабирование с масштабом 6 десятичных позиций. При этом число 1000000 будет выводиться в форме ␣␣1, а последовательность 123 преобразовывается при вводе в число 123000000.

Целое с фиксированным знаком. Если ⟨поле знака⟩ представлено ⟨знаком⟩ (а не ⟨полем плавающего знака⟩), то такой трафарет задает последовательность в форме целого со знаком, находящимся в фиксированной позиции. Если не учитывать возможные редакционные вставки и отбрасывание цифр, то размер последовательности равен $wd+wz+1$. Литера, представляющая знак, находится в начальной позиции. Например, число 12 по трафарету *+4zd* представляется в форме +␣␣␣12.

Заполнение поля цифр определяется в зависимости от трафарета таким же образом, как и для целого без знака.

Если ⟨знак⟩ есть +(—), то для положительных чисел в знаковой позиции находится литера +(пробел), а для отрицательных — литера —(—). Например, число —2 по трафарету *"x="+4zd* представляется в форме *x=*—␣␣␣2, а число 2 — в форме *x=*+␣␣␣␣2.

Целое с плавающим знаком. Наличие в трафарете компоненты ⟨поле плавающего знака⟩, как и в предыдущем случае, задает последовательность литер в форме целого со знаком. Однако здесь знак помещается не в фиксированную позицию, а в одну из позиций поля знака. Размер этого поля равен $we+1$, где we — сумма размеров подполей плавающего знака. В позициях этого поля незначащие нули заменены на пробелы (при вводе и при выводе), а символ, представляющий знак, помещен перед первой значащей цифрой. Например, число —2 по трафарету *"x="4f+d* представляется в форме *x=*␣␣␣␣—2, а число —102 — в форме *x=*␣␣—102.

Если первая значащая цифра находится вне поля знака, то знак помещается в последнюю позицию поля. Например, число —2 по трафарету *'x='3f+2d* представляется в форме *x=*␣␣␣—02.

Заполнение позиций поля цифр определяется в зависимости от трафарета таким же образом, как и для целого без знака. На-

пример, число —2 по трафарету '*x*='3*f*+*zd* представляется в форме *x*=␣␣␣␣—␣2.

Компонента ⟨знак⟩ употребляется в ⟨поле плавающего знака⟩ с тем же смыслом, что и для случая фиксированного знака.

Внутри ⟨поля плавающего знака⟩ может быть задана ⟨условная вставка⟩. Если первая значащая цифра встречается до позиции, с которой начинается вставка, то символы вставки помещаются в строку. В противном случае вместо вставки в строку помещается число пробелов, равное размеру вставки. Если первая значащая цифра непосредственно следует за вставкой, то вставка также заменяется пробелами, но в последнюю позицию поля вставки помещен символ, представляющий знак числа, например, число 100000 по трафарету 4*fc*','*f*+*d* представляется в форме +100,000, число —1000 — в форме ␣␣—1,000, а число 100 — в форме ␣␣␣␣+100.

Заполнение поля цифр определяется в зависимости от трафарета таким же образом, как и для целого без знака. Например число —2 по трафарету 4*fc*','+*zd* представляется в форме ␣␣␣␣—␣2.

Общий размер вводимой/выводимой последовательности литер равен *wt*+*wd*+*wz*+*wi*+*wc*, где *wt* — размер поля знака, равный 0 (если поле знака отсутствует), или 1 (в случае фиксированного знака), или *wf*+1 (в случае плавающего знака); *wi* — суммарный размер безусловных вставок; *wc* — суммарный размер условных вставок.

Число значащих цифр выводимого целого не должно превышать *wd*+*wz*. По ⟨трафарету целого⟩ без знака можно выводить только неотрицательные числа.

Трафарет вещественного. ⟨Трафарет вещественного⟩ предназначен для ввода/вывода последовательности литер, имеющей одну из трех форм вещественного числа:

TU.V (число с фиксированной точкой)
TU.V*e*TP (число с плавающей точкой)
TU*e*TP (число с плавающей точкой)

Здесь T — поле знака (которое может отсутствовать), U — поле цифр целой части, V — поле цифр дробной части, *e* — символ порядка, P — поле цифр порядка.

По ⟨трафарету вещественного⟩ можно выводить вещественные и целые числа.

⟨трафарет вещественного⟩ ::=
⟨трафарет с фиксированной точкой⟩
| ⟨трафарет с плавающей точкой⟩
⟨трафарет с фиксированной точкой⟩ ::=
⟨трафарет целого⟩⟨поле точки⟩⟨поле цифр⟩

⟨трафарет с плавающей точкой⟩ ::=
⟨трафарет с фиксированной точкой⟩ ⟨поле порядка⟩
| ⟨трафарет целого⟩ ⟨поле порядка⟩
⟨поле точки⟩ ::= {*s*}. {⟨вставка⟩}
⟨поле порядка⟩ ::= {*s*} *e* ⟨трафарет целого⟩

Число с фиксированной точкой. Компонента ⟨трафарет целого⟩ предназначена для ввода/вывода знака и цифры целой части числа. Соответствующие позиции обрабатываются по правилам, описанным в разделе трафарет целого.

Компонента ⟨поле цифр⟩ предназначена для ввода/вывода цифр дробной части числа. Позиции этой части заполняются по правилам, описанным для трафарета целого в случае целого без знака. Единственное отличие состоит в том, что если поле цифр целой части заканчивается подполем с *z*-спецификацией, а поле цифр дробной части начинается с такого подполя, то эти подполя объединяются в смысле замены незначащих нулей на пробелы. Т. е. если первая значащая цифра встретилась в подполе целой части, то после нее (в частности, в подполе дробной части) помещаются цифры числа. Если первая значащая цифра встретилась в подполе дробной части, то незначащие нули в целой и дробной части заменяются на пробелы. Если же поле цифр целой части заканчивается подполем с *d*-спецификацией, то поле цифр дробной части рассматривается как независимое целое число.

Компонента ⟨поле точки⟩, во-первых, задает позицию десятичной точки для выравнивания числа при выводе, во-вторых, определяет, каким символом разделены целая и дробная часть. Если в ⟨поле точки⟩ опущена *s*-спецификация, то число представлено в обычном виде, т. е. целая и дробная часть разделены символом точки. Например, число 123.456 по трафарету *3f*+*d.3d* представляется в форме +123.456, а число 0.123 — в форме ␣␣␣+0.123.

Если в ⟨поле точки⟩ указана *s*-спецификация, то оно только задает положение подразумеваемой десятичной точки для выравнивания. Символ точки в строку не помещается. Например, с помощью трафарета "*x*="*sds.3sd3d* можно задать масштабирование. Число 0.000123 по этому трафарету представляется в форме *x*=123.

Трафарет *d*", "*s.2d* позволяет десятичную точку представить литерой «,». Например, число 3,14 представляется в форме 3,14.

Общий размер вводимой/выводимой последовательности равен *wt*+*wd*+*wz*+*wi*+*wc*+1—*ws*, где *ws* — число отбрасываемых литер (цифр и/или символа точки), а остальные обозначения те же, что использованы при описании ⟨трафарета целого⟩. Количество значащих цифр целой части выводимого числа не должно превышать размера поля цифр, задаваемого ⟨трафаретом целого⟩.

Число с плавающей точкой. Компоненты ⟨трафарет с фиксированной точкой⟩ и ⟨трафарет целого⟩ предназначены для ввода/вывода десятичной мантиссы числа с плавающей точкой. В случае ⟨трафарета целого⟩ подразумеваемая десятичная точка находится справа от последней цифры мантиссы. При выводе число выравнивается относительно десятичной точки так, чтобы первая цифра целой части мантиссы была отлична от нуля. Порядок, получившийся в результате выравнивания, выводится в поле цифр порядка. При вводе не требуется, чтобы первая цифра мантиссы была ненулевая. Мантисса обрабатывается по правилам, описанным для числа с фиксированной точкой или для целого, и затем домножается на десять в степени, равной порядку.

Компонента ⟨трафарет целого⟩ в ⟨поле порядка⟩ предназначена для ввода/вывода порядка числа. Эта часть строки обрабатывается как независимое целое число по правилам, описанным для ⟨трафарета целого⟩.

Наличие *s*-спецификации в ⟨поле порядка⟩ означает, что символ порядка не отделяет поле мантиссы от поля порядка.

Например, число 123,456 по трафарету *d.2de*+2*d* представляется строкой 1.23*e*+02, а по трафарету *d.2dse*+2*d* — строкой 1.23+02. Используя в ⟨поле порядка⟩ комбинации из *s*-спецификации и ⟨вставки⟩, можно поле мантиссы отделить от поля порядка любым другим подходящим символом или последовательностью символов.

Общий размер вводимой/выводимой строки равен *wm*+ +*wp*+*wip*+*we*, где *wm* и *wp* — размер поля мантиссы и поля порядка, определяемый так же, как для чисел с фиксированной точкой и целых чисел; *wip* — размер вставки в ⟨поле порядка⟩; *we* равно 1, если *s*-спецификация в ⟨поле порядка⟩ опущена, и равно 0 в противном случае. Абсолютная величина порядка выводимого числа не должна превышать величину, допустимую для поля порядка. В противном случае возникает «ошибка значения».

Трафарет логического. ⟨Трафарет логического⟩ предназначен для ввода/вывода литеры, обозначающей логическое значение (*и* — **истина**, *л* — **ложь**)

⟨трафарет логического⟩ ::= *b* {⟨вставка⟩}

При выводе допускаются наборы, величина которых равна 0 (**ложь**) или 1 (**истина**). При вводе литера преобразуется в логическое значение.

Трафарет литерного. ⟨Трафарет литерного⟩ предназначен для ввода/вывода последовательности литер.

⟨трафарет литерного⟩ ::= ⟨подполе литер⟩...
⟨подполе литер⟩ ::= {⟨повторитель⟩}{*s*}*a*{⟨вставка⟩}

Трафарет *Ra*, где *R* — ⟨повторитель⟩, задает ввод или вывод очередных *R* литер. Трафарет *Rsa* задает пропуск очередных *R* литер выводимого значения; при вводе в результирующую последовательность вставляется *R* пробелов; в обоих случаях текущая позиция по файлу не сдвигается. ⟨Вставка⟩ обрабатывается обычным образом. В общем случае позиция по файлу сдвигается на *wi*+*wa* литер, где *wi* — суммарный размер вставок, *wa* — суммарное число литер, обработанных по трафарету *Ra*. Если необходимо, осуществляется переход на следующую строку.

При выводе допускается набор или вектор элементов литерного формата. Число элементов в обоих случаях должно быть равно *wa*+*ws*, где *ws* — число литер, обрабатываемых по трафарету *Rsa*. В противном случае возникает «ошибка значения». Литеры обрабатываются слева направо.

При вводе допускается переменная или вектор элементов литерного формата. В первом случае *wa*+*ws* литер преобразуются в набор, и набор присваивается переменной. Во втором случае *wa*+*ws* литер присваиваются соответствующим элементам вектора. Длина вектора должна быть равна *wa*+*ws*, иначе возникает «ошибка значения».

13.5. Ошибки форматного обмена. В этом пункте описываются стандартные реакции системы на ошибки форматного обмена.

Ошибка значения. При выводе возможны следующие случаи:

— встречено значение недопустимого типа или числовая величина значения такова, что оно не может быть выведено под управлением заданного формата. В этом случае выводится последовательность литер «*». При выводе под управлением формата длина последовательности определяется этим форматом; при выводе, управляемом данными, длина равна 20 литерам (т. е. размеру целого формата ф64);

— длина выводимого вектора (или набора) отличается от длины, заданной ⟨форматом литерного⟩. Если длина меньше заданной, то к выводимой последовательности справа добавляется нужное число литер «*», в противном случае выводится только начальная часть вектора (или набора) заданной длины.

При вводе возможны следующие случаи:

— направильный тип вводимого значения или абсолютная величина вводимого числа превышает максимально возможную для той переменной, в которую оно вводится. В этом случае переменной присваивается значение, максимально возможное для формата данной переменной;

— длина последовательности литер, вводимой по ⟨трафарету литерного⟩, больше формата переменной или длины вектора, в

которых происходит ввод. В этом случае введенная последовательность обрезается справа до нужной длины.

Во всех случаях выдается сообщение об ошибке.

Ошибка литеры. При возникновении ошибок этого класса выдается соответствующее сообщение.

Значением ⟨форматного обмена⟩ является набор, в котором значения элементов, соответствующих номерам ошибок, установлены в 1.

14. Групповые операции над векторами

В этом разделе описываются групповые операции над векторами элементов простого и строчного форматов: различные варианты пересылок, поиск элемента, удовлетворяющего заданному условию, операции, связанные с переводом чисел из двоичного представления в шестнадцатеричное и обратно, а также из литерного представления в шестнадцатеричное и обратно (упаковка и распаковка). Кроме того, ⟨пересылка⟩ выполняет групповое присваивание значений элементов одной структуры или массива произвольного типа элементом другой структуры или массива эквивалентного типа.

⟨операции над массивами⟩ ::= ⟨пересылка⟩
| ⟨поиск⟩ | ⟨упаковка⟩

Здесь же рассматривается ⟨сравнение строк⟩.

Каждая из этих операций имеет в общем случае несколько операндов. В операциях пересылки значениями основных операндов являются вектор назначения и вектор источника; в операциях поиска и упаковки значением основного операнда является вектор источника. Кроме того, в операциях можно задавать максимальное количество обрабатываемых элементов, эталон для сравнения и другие параметры.

Операции осуществляют циклическую обработку элементов векторов. На каждом шаге выполняются следующие действия: проверка условий окончания операции, обработка текущих элементов, переход к следующему элементу. В связи с тем, что существует несколько условий окончания операции — исчерпание вектора источника, исчерпание вектора назначения, исчерпание заданного числа элементов, выполнение (или невыполнение) заданного отношения,— введены так называемые признаки, принимающие логические значения. По конечному состоянию признаков можно выяснить конкретные обстоятельства, имевшие место в момент окончания операции. В общем виде базовый алгоритм групповых операций выглядит следующим образом:

начало
перем *x*:= *назначение*, *y*:= *источник*,
к#*унцел*:= *максколич*#*унцел*;

```
. . .установить признаки в ложь. . .;
до исчерпвект, исчерпмакс, отношение
цикл
  если к = 0 то исчерпмакс!
  инес длина x = 0 или длина y = 0 то
    исчерпвект!
  иначе
  ...обработка y [0]
  ...присваивание результата обработки переменной x[0]
  x:=x [1:]; y:=y [1:]; к:=к—1
  все
повторить
при
исчерпвект:
  если длина x = 0 то
  . . .установить признак переполнения тгп в истину. . .
  все;
  если длина y = 0 то
  . . .установить признак источника тги в истину. . .
  все,
отношение: . . .установить признак отношения тго в истину. . .
всесит;
  % модифицировать параметры:
...назначение := x; источник := y; максколич:= к ...
...выдать результат...
конец
```

В отношении базового алгоритма следует сделать ряд уточнений.

1. Способ обработки источника, выдаваемый результат и обстоятельства, при которых возникает ситуация *отношение*, зависят от конкретной операции (см. пп. 14.1, 14.2).

2. Максимальное количество обрабатываемых элементов можно не задавать. В этом случае все действия, связанные с количеством, пропускаются. Аналогичным образом трактуется случай отрицательного количества.

3. В операциях поиска и упаковки вектор назначения не участвует. В этих случаях все действия, связанные с ним, пропускаются.

4. Заключительная модификация параметров производится только в том случае, если это явно указано в операции. Модификация задается путем добавления к параметру служебного слова **мод**.

5. Если тип операнда отличен от динамического, то в базовом алгоритме надо заменить соответствующее описание переменной на описание переменной, имеющей тип операнда.

6. Алгоритм выполнения группового присваивания в общем случае отличается от приведенного общего алгоритма.

⟨Операция над массивами⟩ может играть роль ⟨формулы⟩. В этом случае ее тип определяется следующим образом:

— если тип ⟨вектора назначения⟩ (⟨указателя источника⟩) в операции ⟨пересылка⟩ (⟨поиск⟩) — динамический, то и тип ⟨формулы⟩ — динамический, а иначе — тип ⟨вектора назначения⟩ (⟨указателя источника⟩);

— типом ⟨упаковки⟩ является тип *наб*.

⟨Сравнение строк⟩ имеет тип *лог*. ⟨Формула⟩ с операцией **дес** имеет тип *наб*, а с операцией **бит** — тип *дцел*.

Формат элементов вектора назначения и формат элементов вектора источника должны совпадать.

Значением источника, если особо не оговорено, может быть:

— вектор;

— в случае строчного формата — набор, интерпретируемый как вектор элементов, формат которых совпадает с форматом элементов вектора назначения;

— в случае простого формата — набор или другой объект простого формата. В этом случае считается, что вектор источника состоит из одного элемента, значением которого является этот объект.

14.1. Пересылка. Возможны следующие способы пересылки: простая пересылка (условная или безусловная), производящая копирование значений элементов объекта источника в элементы объекта назначения, циклическое заполнение элементов вектора назначения одинаковым содержимым, пересылка литер с перекодировкой, пересылка шестнадцатеричных цифр с распаковкой в литеры. Способ пересылки задается формой ⟨источника пересылки⟩.

```
⟨пересылка⟩ ::=
   ⟨вектор назначения⟩ ⟨операция пересылки⟩
   ⟨источник пересылки⟩
   {&&⟨источник пересылки⟩} ...
⟨источник пересылки⟩ ::=
      ⟨безусловная пересылка⟩ | ⟨условная пересылка⟩
   | ⟨заполнение⟩
   | ⟨перевод⟩ | ⟨распаковка⟩
⟨вектор назначения⟩ ::= ⟨формула⟩ | мод ⟨переменная⟩
⟨операция пересылки⟩ ::= <:=
```

С помощью сцепления ⟨источников пересылки⟩ можно задать последовательность пересылок из различных источников в один вектор назначения. На каждом последующем этапе заполняется группа элементов вектора назначения, непосредственно следующая

за группой, заполненной на предыдущем этапе. Сцепление обозначается парой символов &&.

Один или несколько параметров пересылки (вектор назначения, вектор источника, максимальное количество) можно передать в операцию «именем». Для этого перед соответствующим параметром добавляется служебное слово **мод**. В этом случае соответствующей переменной присваивается модифицированное значение параметра (остаток вектора назначения или источника, количество элементов, оставшихся необработанными). Параметры ⟨источника пересылки⟩ модифицируются по окончании пересылки из этого источника. Переменная, задающая вектор назначения, модифицируется в конце операции.

Если ⟨пересылка⟩ играет роль ⟨формулы⟩, то в качестве результата она выдает остаток вектора назначения.

Простая пересылка. Простая безусловная пересылка заканчивается при исчерпании вектора назначения и/или вектора источника и/или максимального количества. Условная пересылка может закончиться либо по тем же причинам, либо из-за несравнения текущего элемента вектора источника с эталоном.

⟨безусловная пересылка⟩ ::=
  {⟨простой формат⟩} ⟨вектор источника⟩
  длиной ⟨максколичество⟩}
⟨условная пересылка⟩ ::=
  ⟨безусловная пересылка⟩ **пока** ⟨индикатор отношения⟩
  ⟨первичное⟩
⟨вектор источника⟩ ::= ⟨первичное⟩ | **мод** ⟨переменная⟩
⟨максколичество⟩ ::= ⟨первичное⟩ | **мод** ⟨переменная⟩
⟨индикатор отношения⟩ ::=
⟨операция сравнения⟩ | **среди** | **не среди**

Безусловная пересылка векторов определена для простых и строчных форматов. В случае простого формата его необходимо указывать явно. Способ обработки состоит в копировании элемента вектора источника в вектор назначения: $x\,[0] := y\,[0]$.

Условная пересылка определена только для строчных форматов (⟨простой формат⟩ в этом случае нельзя указывать). Способ обработки определяется следующим условным предложением:

если $y[0]$ *R эталон* **то** $x[0]: = y[0]$ **иначе** *отношение*! **все**

где R обозначает ⟨индикатор отношения⟩, а *эталон* — это значение ⟨первичного⟩.

Групповое присваивание произвольных составных объектов имеет вид:

$x <:= y$

где x — ⟨вектор назначения⟩, y — ⟨вектор источника⟩. В этом случае типы x и y должны быть статически связанными эквивалентными типами структур или массивов, причем значения соответствующих параметров типов должны быть попарно равны, а тип x должен быть типом объекта с переменными элементами *). Групповое присваивание состоит в том, что в соответствии с правилами копирования (см. п. 4.1) значениями переменных элементов объекта x становятся значения соответствующих элементов объекта y.

■ *Заполнение.* ⟨Заполнение⟩ присваивает элементам вектора назначения одно и то же значение: $x[0]:=y$, где y является значением ⟨первичного⟩.

⟨заполнение⟩ ::=
{⟨простой формат⟩**}** **заполн** ⟨первичное⟩
{длиной ⟨макскоЛичество⟩**}**

Операция заполнения определена для простых и строчных форматов. В случае простого формата его необходимо указывать явно.

Перевод. ⟨Перевод⟩ пересылает элементы вектора источника с промежуточной перекодировкой: $x[0]:=t[y[0]]$, где t — таблица перекодировки (задается значением ⟨первичного⟩).

⟨перевод⟩ ::=
перевод ⟨вектор источник⟩ **{длиной** ⟨макскоЛичество⟩**}**
по ⟨первичное⟩

Пересылка с переводом определена для литерного формата. Таблица перекодировки также должна быть литерным вектором.

Распаковка. ⟨Распаковка⟩ преобразует шестнадцатеричные цифры источника в соответствующие литеры и пересылает их в вектор назначения.

⟨распаковка⟩ ::=
⟨операция распаковки⟩⟨первичное⟩
{длиной ⟨макскоЛичество⟩**}**
⟨операция распаковки⟩ ::= **распак** | **распакзн**

Значение ⟨первичного⟩ рассматривается как последовательность шестнадцатеричных цифр. Вектор назначения может быть не только литерным, но и цифровым. В последнем случае цифры источника пересылаются без преобразования.

---

*) Составным объектом с переменными элементами называется структура или массив, каждый элемент которого, кроме параметров, является либо переменной, либо встроенной константой с переменными элементами.

Преобразование цифр в литеры состоит в том, что к каждой цифре *z* слева добавляется шестнадцатеричный код *f*. В результате образуется соответствующая цифре литера 4"*fz*".

Операция **распак** выполняет пересылку без учета знака, а операция **распакзн** дополнительно вставляет знак, исходя из текущего значения признака отношения. При распаковке в литеры знак кодируется в поле зоны последней распакованной цифры: 4"*cz*" — положительный знак (**тго = истина**), 4"*dz*" — отрицательный знак (**тго = ложь**). При пересылке в цифровой вектор в качестве первой цифры пересылается знак: 4"*c*" — плюс, 4"*d*" — минус. В этом случае длина результирующей цифровой строки увеличивается на 1. В параметре, задающем максимальное количество пересылаемых цифр, дополнительный элемент не учитывается.

14.2. Операции поиска. Операция ⟨поиска по строке⟩ выполняет поиск элемента строчного формата, удовлетворяющего заданному отношению. Кроме того, существует операция поиска с маскированием (см. гл. 6).

⟨поиск⟩ ::= ⟨поиск по строке⟩ | ⟨поиск по маске⟩

Поиск по строке. Эта конструкция выполняет либо поиск элемента, удовлетворяющего заданному отношению, либо поиск элемента с заданным номером, либо поиск элемента, удовлетворяющего одному из этих условий. В случае проверки отношения способ обработки задается следующим условным предложением:

если *y*[0] *R эталон* **то** *отношение*! **все**

Здесь использованы те же обозначения, что и в п. 14.1. Поиск может прекратиться при обнаружении искомого элемента или при исчерпании вектора источника.

⟨поиск по строке⟩ ::= ⟨указатель источника⟩ **от** ⟨условие поиска⟩
⟨условие поиска⟩ ::=
⟨индикатор отношения⟩⟨первичное⟩
| ⟨максколичество⟩{**либо**⟨индикатор отношения⟩
⟨первичное⟩}
⟨указатель источника⟩ ::= ⟨формула⟩ | **мод** ⟨переменная⟩

Эталон для сравнения задается значением ⟨первичного⟩, а номер искомого элемента компонентой ⟨максколичество⟩. Операция определена для векторов, состоящих из элементов строчного формата. Значением ⟨указателя источника⟩ должен быть вектор, остальные типы в данном случае запрещены. Если ⟨поиск по строке⟩ играет роль ⟨формулы⟩, то результатом операции является остаток вектора источника. В случае, если операция закончилась не из-за исчерпания источника, начальным элементом результирующего вектора является искомый элемент.

Если в конструкции предписана модификация параметра, задающего номер искомого элемента, необходимо учитывать, что в соответствии с базовым алгоритмом модифицированное значение определяет разницу между первоначальным значением и числом пропущенных элементов, а не номер найденного элемента.

14.3. Сравнение строк. Операция ⟨сравнения строк⟩ производит поэлементное сравнение векторов, состоящих из элементов строчного формата, и выдает логическое значение **истина** или **ложь**. Операция продолжается до исчерпания одного из векторов.

В случае литерных и цифровых строк результатом операции является результат сравнения первой встретившейся пары неравных элементов. При сравнении битовых строк значение **истина** выдается только в том случае, если заданное соотношение имеет место для всех сравниваемых пар. Это свойство имеет теоретико-множественное толкование: если строки x и y задают подмножества некоторого множества (элементы строки, соответствующие элементам подмножества, имеют значение **истина**, а остальные — **ложь**), то $x=/y$ проверяет эквивалентность подмножеств, $x<=/y$ проверяет включение x в y, а $x>=/y$ проверяет включение y в x.

⟨сравнение строк⟩ ::= ⟨вектор источника⟩

⟨операция сравнения⟩ / ⟨вектор источника⟩
{длиной ⟨максколичество⟩**}**

⟨операция сравнения⟩ ::= < | > | = | < > | >= | <=

Правый операнд может принимать такие же значения, как источник в операциях пересылки. Значением левого операнда должен быть указатель вектора. Установка признаков и заключительная модификация параметров имеют такую же трактовку, как и в других групповых операциях. По завершении операции состояние признака отношения совпадает с результатом операции.

В случае, если длина какого-либо из исходных векторов и/или начальное значение ⟨максколичества⟩ оказались равными нулю, векторы считаются равными.

14.4. Операции перевода чисел. Значение операнда операции дес должно принадлежать типу *унцел*. Операция переводит целое число в десятичное представление. Результатом является набор, интерпретируемый как цифровой. Элемент формата ф4 с номером k равен десятичной цифре, находящейся в k-й позиции. Длина набора (в терминах элементов формата ф4) равна количеству значащих цифр в переведенном числе. Если количество цифр превышает 16, то признак переполнения **тгп** устанавливается в 1, иначе — в 0. Если исходное число имеет отрицательный знак, признак отношения **тго** устанавливается в 1 (**истина**), иначе — в 0 (**ложь**).

Значение операнда операции **бит** должно быть набором, который интерпретируется как цифровой набор. Операция осуществ-

ляет обратный перевод из десятичного представления в двоичное. Результатом является целое типа *дцел* (в зависимости от исходного состояния признака отношения результату приписывается положительный или отрицательный знак).

14.5. Упаковка. Операция упаковки предназначена в основном для преобразования целого числа из литерного представления в набор шестнадцатеричных цифр.

⟨упаковка⟩ ::=
пак ⟨вектор источника⟩{**длиной** ⟨максколичество⟩}

Значением ⟨вектора источника⟩ может быть либо вектор элементов формата ф4 или ф8, либо набор, интерпретируемый как вектор элементов литерного формата. Преобразование литер состоит в том, что в каждом элементе отбрасывается «зона» — левые четыре разряда. Шестнадцатеричные цифры переносятся в результирующий набор без преобразования. Результатом операции является набор, в котором количество элементов цифрового формата равно количеству обработанных элементов исходного вектора. Если количество превышает число 16, то возникает ситуация *неверныйоперанд*.

Обрабатываемая строка может содержать знак числа, закодированный в такой форме, как это описано в операции ⟨распаковки⟩ **распакзн**. В этом случае операция упаковки производит обратное преобразование и восстанавливает состояние признака отношения. Знаковый элемент исходной цифровой строки не учитывается в параметре ⟨максколичество⟩.

14.6. Признаки. Значение ⟨признаков⟩ устанавливается и анализируется в процессе выполнения операций перевода чисел и групповых операций над векторами. По текущему значению признаков можно судить о том, какие обстоятельства привели к окончанию групповой операции. ⟨Признак⟩ может иметь одно из логических значений: **истина** или **ложь**.

⟨признак⟩ ::= **тго** | **тгп** | **тги**

Служебные слова имеют следующий смысл: **тго** — признак отношения, **тгп** — признак переполнения вектора назначения, **тги** — признак исчерпания источника.

Явным образом можно установить только признак отношения. Это делается с помощью стандартной функции *тгз*; *тгз* (*x*) устанавливает признак отношения в 1 (**истина**), если значение *x* — отрицательное число, и в 0 (**ложь**) — в противном случае.

Перед инициированием выполнения базированного блока текущие значения признаков запоминаются, а по окончании выполнения восстанавливаются.

15. Преобразование формы массива

В этом разделе описываются средства языка, позволяющие из элементов одного массива составить другой массив, форма которого (число измерений, длина измерений и пр.) отличается от исходной. Следует отметить, что при этом дополнительная память не выделяется и элементы первого массива не копируются, а лишь формируется паспорт, по-новому описывающий нужную часть исходного массива. Таким образом, составление нового массива заключается в наложении нового паспорта на исходный массив. В связи с этим новый массив в этом разделе называется наложенным массивом. Исходный паспорт в общем случае сохраняется. По этой причине одновременно могут существовать несколько паспортов, различным образом описывающих один и тот же массив.

15.1. Формирование паспорта. Конструкция ⟨формирование паспорта⟩ вводит наложенный массив, который образуется из исходного путем следующих основных преобразований: сдвиг начала измерений, сокращение длин измерений, увеличение шага по измерениям (растяжение координатных осей), сокращение числа измерений, перестановка измерений местами.

⟨формирование паспорта⟩ ::=
формавм ({⟨описатель⟩}[⟨список идентификаторов⟩] **=**
⟨описатель⟩ [⟨список диапазонов или индексов⟩])
⟨описатель⟩ ::= ⟨идентификатор⟩ | ⟨закрытое выражение⟩
⟨диапазон или индекс⟩ ::= ⟨суперпозиция⟩ {**=** ⟨диапазон⟩}
⟨суперпозиция⟩ ::= ⟨слагаемое⟩{⟨знак⟩ ⟨слагаемое⟩}...
⟨слагаемое⟩ ::= ⟨сомножитель⟩ {* ⟨сомножитель⟩}

Исходный массив задается значением ⟨описателя⟩, находящегося в правой части. Значение ⟨описателя⟩ в левой части задает объект типа паспорт. В этот объект копируется вновь сформированный паспорт. В целом смысл конструкции состоит в том, что она задает соответствие между элементами наложенного и исходного массивов. При этом используется метод, схожий с методом формальных параметров. ⟨Список идентификаторов⟩ в левой части действует аналогично объявлению формальных параметров. В нем вводятся идентификаторы, с помощью которых в правой части обозначаются индексы наложенного массива.

В каждой индексной позиции правой части может находиться линейная ⟨суперпозиция⟩ формальных индексов. В этом случае ⟨идентификаторы⟩ формальных индексов являются ⟨сомножителями⟩. Например:

формавм ($a[i] = b[2 * i]$)

где i — формальный индекс, означает, что элементы наложенного массива a — это четные элементы исходного массива b.

Чтобы определить элемент исходного массива, соответствующий конкретному элементу наложенного массива, надо индексы последнего подставить как фактические параметры в правую часть. Получившиеся при этом индексы по исходному массиву не должны выходить за разрешенный диапазон. Нижняя граница задается значением первого ⟨выражения⟩ в ⟨диапазоне⟩, а длина диапазона — значением второго ⟨выражения⟩. Если первое (второе) ⟨выражение⟩ опущено, то подразумевается, что нижняя (верхняя) граница совпадает с нижней (верхней) границей исходного массива по данному измерению. ⟨Диапазон⟩ может быть полностью опущен вместе со знаком равенства. Это означает, что разрешенный диапазон совпадает с диапазоном изменения индекса исходного массива по данному измерению.

Если в индексной позиции находится ⟨суперпозиция⟩, не зависящая от формальных индексов, то тем самым индекс по данному измерению исходного массива фиксируется и полагается равным значению ⟨суперпозиции⟩. При этом число измерений, очевидно, уменьшается на 1. В данном случае ⟨диапазон⟩ не задается. Например,

формавм $(a[i] = bb[2 * i,\ 0])$

означает, что элементы наложенного вектора a — это четные элементы нулевого столбца исходной матрицы bb.

Число измерений наложенного массива не должно превышать числа измерений исходного массива. Каждый формальный индекс должен входить по крайней мере в одну ⟨суперпозицию⟩, но может встречаться и в нескольких.

⟨Идентификатор⟩ формального индекса может находиться в правой части ⟨формирования паспорта⟩ только на месте ⟨сомножителя⟩ ⟨суперпозиции⟩. Причем, если один ⟨сомножитель⟩ какого-либо ⟨слагаемого⟩ — формальный индекс, то другой ⟨сомножитель⟩ не может быть таковым, т. е. ⟨суперпозиция⟩ линейно зависит от формальных индексов.

Компоненты ⟨формирования паспорта⟩, отличные от ⟨идентификаторов⟩ формальных индексов, выполняются в момент выполнения всей конструкции. Значением ⟨описателя⟩ в левой части является объект типа паспорт n-мерного массива, где n — число формальных индексов. Если левый ⟨описатель⟩ опущен, то такой объект создается неявно и локализуется в ближайшем блоке.

Значением конструкции является наложенный массив. При заданном левом ⟨описателе⟩ конструкцию можно использовать в роли ⟨оператора⟩.

Случай параллелепипеда. Пусть в левой части вводится n формальных индексов, а каждый (диапазон или индекс) в правой части имеет вид $m*i+c=к$: $к$, где i — один из формальных индексов ($к$, m, c и $к$ могут быть опущены; если члену суперпозиции предшествует отрицательный знак, то он учтен в m или c). Тогда элементы наложенного массива составляют в n-мерном координатном пространстве прямоугольный параллелепипед *).

Пример.

```
конст
    ат = формавм ([i,j] = a[j,i]),  % ат — паспорт транспониро-
                                    % ванной матрицы a
  адиаг = формавм ([i] = a[i, i]),  % адиаг — паспорт диаго-
                                    % нали матрицы a
  аобрдиаг = формавм ([i] = a[i, (l—1)—i]),  % аобрдиаг—паспорт
                                    % обратной диаго-
                                    % нали матрицы a
                                    % размером l*l
  минор = формавм ([i,j] = a[i =0 : к, j = 0 : к])
                                    % паспорт минора
                                    % к-го порядка.
```

Индекс i может пробегать следующий диапазон значений:

$$(к—c)/:m \leqslant i \leqslant (к+к—1—c)/:m$$

Если i встречается в нескольких суперпозициях, таких неравенств несколько, и диапазон изменения индекса определяется путем выбора минимальной верхней границы и максимальной нижней. Результирующая нижняя граница должна быть не больше верхней.

Как видно из предыдущего, нижняя граница индексов наложенного массива может оказаться отличной от нуля, и, в частности, — отрицательной. Язык допускает индексацию таких в-массивов, но средства, позволяющие по паспорту установить нижние границы, отсутствуют. Однако, используя атрибут *нигрнуль*, можно установить все нижние границы в нуль, и обрабатывать его обычным образом, предполагая, что каждый индекс изменяется, начиная от нуля в сторону положительных значений.

Паспорт прямоугольного параллелепипеда обладает следующими свойствами:

1. Для него определены значения всех атрибутов паспорта. В частности, путем запроса атрибута *длиниэм* можно определить длины по измерениям.

2. К нему применимы все конструкции и операции, связанные с в-массивами.

---

*) Обычный в-массив, создаваемый с помощью генератора в-массива, всегда является параллелепипедом.

Случай произвольной суперпозиции. Это случай, когда по крайней мере в одну из суперпозиций входит более одного формального индекса. Такое наложение используется лишь с целью оптимизации. Оно позволяет любую индексацию массива внутри цикла представить в нормальной форме. Нормальной формой называется индексация вида $a\ [i1, i2, \ldots, in]$, где a — идентификатор константы или переменной, а $i1, i2, \ldots, in$ — параметры циклов. В этом случае транслятор обеспечивает генерацию наиболее эффективного кода.

Пример.

Во вложенном цикле происходит обращение к элементам массива *a*:

```
для i до ni
цикл
  для j до nj
  цикл
    ... a [2 * i + j, j + 1] ...
  повторить
повторить
```

Используя ⟨формирование паспорта⟩, можно перейти к индексации в нормальной форме:

```
конст aonm = формавм ([i1, j1] = a [2 * i1 + j1, j1 + 1]);
для i до ni
цикл
  для j до nj
  цикл
    ... aonm [i, j]
  повторить
повторить
```

В случае произвольной суперпозиции особенность результирующего паспорта состоит в следующем:

— для него не определен ряд атрибутов,

— его нельзя использовать в правой части конструкции ⟨формирование паспорта⟩,

— для него не определены стандартные функции над паспортами (см. ниже).

15.2. Стандартная функция *пввект.* Значением функции *пввект* (*a, b, bu*) является выстроенный вектор, наложенный на элементы n-мерного массива $b[i_1, i_2, \ldots, i_{n-1}, i_n]$, где индексы $i_1, i_2, \ldots, i_{n-1}$ фиксированы, а i_n пробегает все допустимые значения. Индексы $i_1, i_2, \ldots, i_n$ являются значениями элементов вектора *bu*, содержащего ровно $n-1$ элемент:

$$bu[0]=i_1,\quad bu[1]=i_2,\quad \ldots,\quad bu[n-2]=i_{n-1}.$$

Параметр *a* задает объект типа паспорт (одномерного массива), куда и копируется результирующий паспорт; он же является результатом функции.

Эта функция в сочетании с атрибутами паспорта *чслизм* (число измерений) и *длинизм* (длины измерений) позволяет обрабатывать выстроенный массив, число измерений которого статически неизвестно.

Функция может вызываться с одним параметром: *ввект* (*b*). В этом случае *b* — паспорт одномерного в-массива. Он же является значением функции. Этот вариант предназначается для того, чтобы квалифицировать значение выражения *b* (в частности, — формальный параметр, см. п. 6.2) как паспорт в-вектора.

15.3. **Генератор паспорта.** ⟨Генератор паспорта⟩ позволяет создать объект типа паспорт. Этот паспорт является подвижным, т. е. его содержимое можно менять таким образом, чтобы он в разное время описывал разные массивы.

⟨генератор паспорта⟩ ::=
⟨локализация⟩ [{,} ...]
| ⟨локализация⟩[⟨список выражений⟩]

Пример.

```
начало
  конст a = лок [,]; % создать паспорт
  . . .
  % наложить паспорт a на транспонированную матрицу b
  формавм (a [i, j] = b [j, i]);
  . . .
  % наложить паспорт a на транспонированную
  % матрицу c:
  формавм (a [i, j] = c [j, i]);
  . . .
конец
```

Значение атрибута *чслизм* созданного паспорта (число измерений, см. атрибуты паспорта) равно количеству индексных позиций между скобками. Если ⟨список выражений⟩ опущен, то длины по измерениям (атрибут *длинизм*) первоначально равны 0. В противном случае (а) они равны значению соответствующих ⟨выражений⟩ и (б) содержимое созданного паспорта можно менять только путем модификации атрибута *базвект*.

Постоянные и подвижные паспорта. Паспорта, создаваемые ⟨генератором массива⟩ и неявно — ⟨формированием паспорта⟩ (при опущенном ⟨описателе⟩ в левой части конструкции), являются постоянными, т. е. они описывают один и тот же массив. Их нельзя накладывать на другой массив с помощью

⟨формирования паспорта⟩ или путем модификации атрибута *базвект*.

В отличие от этого содержимое паспорта, созданного с помощью ⟨генератора паспорта⟩, изменять можно (с учетом указанного выше ограничения).

15.4. Типы операндов. Типом ⟨генератора паспорта⟩ является тип паспорта *стпаспвм* (*n*), где *n* — число индексных позиций. В конструкции ⟨формирование паспорта⟩ значением левого ⟨описателя⟩ должен быть объект типа *стпаспвм* (*n*), где *n* — число индексных позиций слева. Если правый ⟨описатель⟩ имеет динамический тип, то в зависимости от числа индексных позиций справа различаются два случая: (1) одна индексная позиция, тогда значением ⟨первичного⟩ должен быть вектор типа *ствект*; (2) *к* индексных позиций (*к*>1), тогда значением первичного должен быть массив типа *ствм*(*к*). Если тип правого ⟨описателя⟩ отличен от динамического, то он должен входить в состав типов *ствект* и *ствм*. Типом конструкции ⟨формирование паспорта⟩ является тип *ствм* (*n*), где *n* — число индексных позиций слева. Типом результата функции *пввект* является тип *вм*1.

Замечание. Если значением *x* является паспорт, который в результате выполнения ⟨формирования паспорта⟩ стал паспортом конкретного *n*-мерного массива, проверка типа *провтип* (*x*, *ствм* (*n*)) дает результат **истина**. Поэтому такой объект можно использовать везде, где требуется тип массива. Если значением *x* является массив типа *ствм* (*к*), полученный в результате сформирования паспорта, проверка типа *провтип* (*x*, *стпаспвм* (*к*)) дает результат **истина**, и соответствующий атрибут паспорта данного массива указывает на то, что паспорт является подвижным. Поэтому такой массив можно использовать в качестве значения левого ⟨описателя⟩ в конструкции ⟨формирование паспорта⟩. Подробнее о проверке типов промежуточных объектов см. гл. 5.

16. Участок базированной области

⟨Описание базы⟩ вводит идентификатор константы, значением которой является вектор стандартного типа *стсв* (*ткор*), элементами которого являются элементы области памяти, содержащей переменную часть параметров процедуры стандартного типа (см. п. 11.1). Выполнение самого этого описания не приводит к распределению дополнительной памяти.

⟨описание базы⟩ ::= **база** ⟨идентификатор⟩

Необходимо учитывать, что элементы вектора имеют формат ф32; при необходимости формат можно преобразовать с помощью конструкции ⟨подмассив⟩ (см. примеры в п. 11).

17. Текстовые макросы

⟨Описание текстов⟩ и ⟨подстановка текста⟩ представляют собой простейшую форму макросредств языка. Описание связывает идентификатор с текстом ⟨предложения⟩, находящегося в правой части. ⟨Подстановка текста⟩ выполняется в процессе трансляции путем копирования текста ⟨предложения⟩ в место подстановки.

⟨описание текстов⟩ ::= **текст** ⟨список идентификации текстов⟩
⟨идентификация текста⟩ ::=
⟨идентификатор⟩ {({⟨параметры текста⟩})}
{#⟨вычисление типа⟩}= ⟨предложение⟩
⟨параметры текста⟩ ::=
⟨список предварительных идентификаций⟩

⟨Предложение⟩ может быть ⟨выражением⟩, ⟨оператором⟩ либо ⟨переменной⟩. Выбор зависит от того, какие синтаксические формы разрешены в тех местах программы, куда подставляется текст данного ⟨предложения⟩. Если указано ⟨вычисление типа⟩, то тем самым задан тип ⟨выражения⟩ или тип ⟨переменной⟩.

Текст может быть параметризован с помощью формальных параметров текста, идентификаторы и типы которых указаны в левой части ⟨идентификации текста⟩. Областью действия формальных параметров является ⟨предложение⟩ в правой части. Каждое вхождение формального параметра внутри ⟨предложения⟩ рассматривается как ⟨подстановка текста⟩ без параметров. Если в заголовке текста не указан тип какого-либо формального параметра, и ⟨идентификатор⟩ этого параметра внутри ⟨предложения⟩ выполняет роль ⟨первичного⟩ или ⟨переменной⟩, то считается, что это ⟨первичное⟩ или ⟨переменная⟩ имеет динамический тип, а иначе ⟨первичное⟩ или ⟨переменная⟩ имеет тип, указанный в заголовке.

⟨подстановка текста⟩ ::=
⟨идентификатор⟩ {({⟨список предложений⟩})}
| ⟨идентификатор⟩ [⟨список предложений⟩]

⟨Подстановка текста⟩ выполняется следующим образом. ⟨Предложение⟩, связанное с ⟨идентификатором⟩ текста, заключается в скобки и копируется в место подстановки. В процессе копирования рекурсивно выполняются все внутренние подстановки текстов. В частности, аналогичным образом выполняется подстановка фактических параметров. В получаемых при этом конструкциях должны выполняться допустимые для этих конструкций соотношения типов операндов. При подстановке текста, как и при вызове процедуры, идентификаторы сохраняют свой смысл: идентификаторы, входящие в ⟨предложение⟩, имеют тот смысл, который им был приписан

в месте описания текста, и идентификаторы, использованные в фактических параметрах, имеют тот смысл, который им был приписан в месте подстановки текста.

Число фактических параметров текста должно совпадать с числом формальных параметров. Тип фактического параметра должен статически принадлежать типу соответствующего формального параметра. Пустые скобки в ⟨подстановке текста⟩ используются в том случае, если они были заданы в левой части описания этого текста. Это позволяет одинаковым образом обозначать ⟨вызов⟩ процедуры и ⟨подстановку текста⟩.

Выполнение ⟨подстановки текста⟩ не должно приводить к рекурсивной подстановке того же самого текста.

18. Атрибуты объектов

Конструкция ⟨запрос атрибута⟩ выдает текущее значение заданного атрибута объекта стандартного скрытого типа, а конструкция ⟨модификация атрибутов⟩ изменяет значение указанных атрибутов.

Атрибутам поставлены в соответствие номера атрибутов — стандартные целочисленные константы, с помощью которых атрибуты обозначаются при запросе и модификации. Семантика атрибутов описана в гл. 4.

⟨запрос атрибута⟩ ::=
читатр (⟨выражение⟩, {⟨уточнение⟩,} ⟨идентификатор⟩)
⟨уточнение⟩ ::= ⟨выражение⟩

Значением первого ⟨выражения⟩ является объект, у которого выполняется запрос атрибута. ⟨Идентификатор⟩ константы задает номер атрибута.

⟨модификация атрибутов⟩ ::=
запатр (⟨выражение⟩, {⟨уточнение⟩,}
⟨список установок атрибутов⟩)
⟨установка атрибута⟩ ::= ⟨идентификатор⟩:⟨выражение⟩

В этой конструкции первое ⟨выражение⟩ трактуется, как и в предыдущей. В списке установки атрибутов перечисляются изменяемые атрибуты: ⟨идентификатор⟩ задает номер атрибута, а значение ⟨выражения⟩ — новое значение атрибута.

⟨Модификация атрибутов⟩ может использоваться и в роли ⟨оператора⟩ и как ⟨выражение⟩. Во втором случае ее значением является значение первого параметра.

В обеих конструкциях может присутствовать компонента ⟨уточнение⟩. Она используется в ряде случаев при запросе атрибутов в-массива (см. § 9, гл. 4), а также при запросе/модификации атри-

бутов внешних объектов и оперативных объектов связи (см. § 13, гл. 3).

Примеры.

```
читатр (а, чслизм)              % число измерений массива
читатр (а, 1, длинизм)          % длина первого измерения
                                % массива а
запатр (ф, типфайла : текст)    % изменение атрибута
                                % типфайла у файла ф
```

Ряд дополнительных сведений о запросе и модификации атрибутов внешних объектов и оперативных объектов связи рассмотрен в § 3 гл. 3.

19. Описание и использование меток

В этом разделе описываются средства языка, предназначенные для программирования с использованием оператора перехода.

⟨Описание метки⟩ является предварительным описанием идентификатора, используемого в качестве ⟨метки⟩ некоторого предложения.

⟨описание меток⟩ ::= **метка** ⟨список идентификаторов⟩

⟨метка⟩ ::= ⟨идентификатор⟩^:

⟨Описание метки⟩ должно предшествовать ⟨метке⟩ и любому другому использованию ⟨идентификатора⟩ метки. ⟨Описание метки⟩ и ⟨метка⟩ должны находиться на одном и том же уровне структурной вложенности предложений программы, т. е. если ⟨описание метки⟩ дано в начале закрытого или последовательного предложения, то соответствующая ⟨метка⟩ не может встречаться внутри вложенного закрытого или последовательного предложения с описаниями. ⟨Идентификатор⟩ метки трактуется как идентификатор константы, значением которой является соответствующая метка перехода.

Оператор перехода передает управление на то предложение, которое помечено меткой, являющейся значением его операнда.

⟨переход⟩ ::= **на** ⟨одноместная формула⟩

При выполнении глобального перехода прекращается выполнение всех промежуточных блоков; попутно уничтожаются локализованные в них объекты. ■

20. Примеры

В этом параграфе приводятся примеры программ. В примерах 1—8 исходные данные задаются в программе, а результаты выводятся на печать. В примере 9 представлено задание, в котором вызывается программа с двумя входными файлами-параметрами.

Пример 1. Работа с циклом. Вычисление суммы ряда,

$h(n) = 1 + 1/2 + 1/3 + \ldots + 1/n$

```
начало
   конст печ = позп (ацпу);
   процедура h = функция (конст к#цел)#вещ
      начало
      перем сум#вещ := минвещ;
         для i от к вниздо 1
         цикл
            сум := сум + 1/i
         повторить;
         сум
      конец;
   запф (печ, h (10): "сумма =" g (0, 13))
конец
```

Пример 2. Выход из цикла по ситуации. Вычисление косинуса.

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots + (-1)^k \frac{x^{2k}}{(2k)!}$$

до $a_k/\cos(x) < eps$, где a_k — k-й член суммы.

```
начало
   конст печ = позп(ацпу);
   процедура cos = функция (конст х#квещ) #квещ
      начало
         конст eps = 1е — 14;
         перем у#вещ := 1.0, ак#вещ := 1.0,
         к#цел := 0;
      до отношмнше
      цикл
         конст хкв#вещ = ф64окр (х * х);
         если абс ак < eps * абс у то отношмнше ! все;
         к : = к + 2;
         ак : = — ак * хкв/(к*(к — 1));
         у ;= у + ак
         повторить;
         запф (печ, к/: 2 : "к = "2zd);   % число итераций
   ф32окр у
   конец;
   запф (печ, 1.57: "cos ("g (0,4)") = ", cos (1.57) : g (0, 4))
конец
```

Пример 3. Выход из цикла по ситуации и со значением. В примере вычисляется $z = u ** e$, где e — целое.

```
начало
   конст печ=позп (ацпу);
   процедура возвстеп = функция (конст и#чис, ве#цел) #чис
      до исчерпстеп
      цикл перем z#чис := 1, v#чис := и, е#цел := ее;
         если е = 0 то исчерпстеп ! (z)
         иначе
            до нгчет
            цикл
             если е остат 2 = 1 то нечет !
             иначе v := v * v
             все;
             e := e/: 2
            повторить;
            z := z * v
         все;
         e :=e — 1
      повторить
      при исчерпстеп (zz#чис) : zz % результат
      всесит;                       % конец возвстеп
   запф (печ, возвстеп (5, 3) :"5 ** 3 = " g (0))
конец
```

Пример 4. Работа с битовыми векторами. Решето Эратосфена.

```
начало
   процедура решето = проц (конст п#цел)
      начало
         конст печ = позп (ацпу),
            реш#свлог = ген свлог (п) иниц ( : истина);
         перем к#цел := 1;
         запф (печ, 1 : 7zd);
         для i от 2 до п — 1
         цикл
            если реш [i] то
               запф (печ, i : 7zd);
               если (к := к + 1) > 16 то %перевод строки
               к := 1;
               запф (печ, : l)
               все ;
               до верхгран
               цикл перем j#цел := i ;
               реш [j] := ложь ;
               если (j := j + i) > (п — 1)
               то верхгран ! все
```

```
            повторить
          все
        повторить
      конец;     % процедуры решето
  решето (10000)
конец
```

Пример 5. Работа с цифровыми и литерными векторами.

```
начало
  процедура отрстеп2 = проц (конст п#цел)
    начало     % печать числа 2 ** (— п)
      конст
        т#свциф = ген свциф (п),
        s#свлит = ген свлит (п),
        печ = позп (ацпу) ;
      для к до п — 1
      цикл     % от 2 ** (—1) до 2 ** (—п)
       % к первых цифр частного
        для i до к — 1
        цикл перем r#дцел := 0 ;
          % алгоритм деления «столбиком»:
          r := 10 * r + т [i];     % остаток от предыд.
                                   % итерации + в хвост
                                   % следующая цифра де-
                                   % лимого
          т [i] := измтип (r/:2, циф); % цифра частного
          r := r остат 2; % остаток
          s [i] := измтип ("0" + т [i], лит);% цифра
                                             % в литеру
        повторить;
        % к + 1-я цифра частного:
        т [к] := 4 "5" ;
        s [к] := "5" ;
        запф (печ, к + 1 : "1/2 ** "2zd " = 0.", s[1 : к + 1],: i)
      повторить
    конец ; % процедуры отрстеп2
  отрстеп2 (10)
конец
```

Пример 6. Сортировка массива методом простой вставки.

```
начало
  тип тм=массив [ ] перем#чис;
  конст печ=позп (ацпу),
    b# тм = ген тм (5) иниц (0:5, 1:3, 2:3, 3:4, 4:0);
  процедура сортпр = проц (конст а # тм)
```

```
      для i от 1 до длина a — 1
      цикл     % слева от i-го элемента
               % ищется меньший или равный
               % a[i]; затем a[i] вставляется перед ним
         перем ai#чис;
         ai := a[i];     % сохранить a[i]
         до найден
            (для j от i — 1 вниздо 0
            цикл
               если a[j] <= ai то найден!(j)
               иначе % сдвинуть j-й вправо
               a[j+1] := a[j]
               все
            повторить;
            найден!(—1))
         при
            найден (к#цел): a[к+1]:=ai
         всесит
      повторить;     % сортпр
   запф (печ, : "исходный массив: ", b: 10g (10));
   сортпр (b);
   запф (печ, : "результирующий массив: ", b: 10g (10))
конец
```

Пример 7. Сортировка массива методом двоичной вставки

```
начало
   тип тм = массив [ ] перем#чис;
   конст печ = поэп(ацпу),
      b#тм = ген тм(5)иниц (0 : 20, 1 : 21, 2 : 16, 3 : 18, 4 : 16)
   процедура сортдв = проц (конст a#тм)
      для i от 1 до длина a — 1
      цикл     % место для i-го элемента ищется
         перем ai#чис ; % методом двоичного поиска
         ai := a [i] ;     % сохранить a [i]
         до пересеч
         цикл
            перем m#цел,     % середина отрезка
            l#цел := 0,      % левая граница
            r#цел := i — 1 ; % правая граница
            если l > r то пересеч ! (l)
            иначе
               m := (l + r) / : 2 ;
               если ai < a [m] то r := m — 1
               иначе l := m + 1
               все
               все
```

```
        повторить
        при пересеч (ll#цел) :
           для j от   i — 1 вниздо ll
           цикл          % сдвинуть элементы
              a [j + 1] := a [j]
           повторить ;
           a [ll] := ai
        всесит
     повторить ; % сортдв
  запф (печ, : "исходный массив :", b : 5g (10)) ;
  сортдв (b) ;
  запф (печ, : "результирующий массив :", b : 5g (10))
конец
```

Пример 8. Рекурсивные процедуры. Быстрая сортировка.

```
начало
  тип тм = массив [ ] перем#чис ;
  конст печ = позп (ацпу),
     b#тм = ген тм(5) иниц (0 : 20, 1 : 21, 2 : 16, 3 : 18, 4 : 16) ;
  процедура быстрсорт = проц (конст a#тм)
     начало
        процедура раздел = проц (конст l#, r#цел)
           начало
           перем i#цел := l, j#цел := n ;
              до указвстретились
              цикл
                 конст x#чис = a [(l + r) /: 2] ;
                 до найденмнш
                 цикл
                    если a [i] > = x то найденмнш !
                    иначе i := i + 1 все
                 повторить;
                 до найденбольш
                 цикл
                    если a [j] < = x то найденбольш !
                    иначе j := j — 1
                    все
                 повторить ;
                 если i < =j то % поменять местами ai и aj
                    конст w#чис = a [i] ;
                    a [i] := a [j] ; a [j] := w ;
                  i := i + 1 ; j := j — 1
                 все ;
                 если i > j то указвстретились! все
              повторить ;
```

```
                если l < j то раздел (l, j) все;
                если i < r то раздел (i, r) все
              % иначе раздел состоит из одного элемента,
              % который находится на своем месте
            конец     % раздел
        раздел (0, длина a — 1)
    конец ;                % быстрсорт
  запф (печ , : "исходный массив : ", b : 10g (10)) ;
  быстрсорт (b);
  запф (печ , : "результирующий массив :" , b : 10g (10))
конец
```

Пример 9. Программа пакетного задания. В примере описывается задание, выполняющее вызов программы, текст которой «вложен» в текст задания.

```
начало
  конст тест = тфайл (имяяз : "авт")
    *!!*
    проц (п, фа, фв)
  начало
    конст а#свкор = ген свкор(п), в#свкор = ген свкор(п),
      печ = позп(ацпу) ;
    процедура скапр = функция (конст х#, у#свкор)#кчис
      (перем сум#чис : = 0 ;
      для i до длина х — 1
      цикл % суммирование с удв. точностью:
        сум : = сум + х [i] умнд у [i]
      повторить ;
      ф32окр сум ; % округление результата
    читф (фа, а) ; читф (фв, в) ; % ввод массивов
    вапф (печ , : "скалярное произведение =",
    скапр (а, в) : g (0, 1)) % печать результата
  конец  % теста
  !!,
  ввв = тфайл
    *!!*
    1 2 3 4
    !!,
  ввв = тфайл
    *!!*
    5 6 7 8
    !! ;
  тест (4, позп (ввв), позп (ввв))
конец     % задания
```

Это же задание можно записать короче, ликвидировав описания констант и изобразив все файлы непосредственно в конструкции ⟨вызов⟩ :

```
начало
   тфайл (имяяз :"авт")
   *!!*
   проц (n, фа, фв)
   начало
      . . .
   конец
   !!
   (4 ,
   поэп (
   тфайл
      *!!*
      1 2 3 4
      !! ) ,
      поэп(
      тфайл
      *!!*
      5 6 7 8
      !!) )
конец
```

Пример 10. Предварительное описание типов. Рекурсивные типы данных.

```
начало
   конст вж = генвж(  );
   тип
   ту = (терм, узел) ,
   двдер = выбтип ту из
      терм: цел,
      узел: структ (конст кодузла#лит,
                           лветвь#, пветвь#двдер)
      всевыб;
   % объект типа двдер может быть либо термом — значением
   % типа цел, либо узлом, который характеризуется
   % кодом бинарной операции и двумя ветвями, определяющим
   % ее операнды.
   процедура
   созддер=функция (конст х#лит,
      лв#, пв#двдер)#двдер
   % процедура создания объекта типа двдер (узел).
      (ген двдер (узел) [локал : вж]
      иниц (кодузла : х, лветвь : лв, пветвь : пв) ) ;
```

```
    перем x#двдер;
    % пример создания дерева, соответствующего выражению:
    % 2+3*5
    x := созддер ("+", 2, созддер ("*", 3, 5))
конец%примера 10.
```

Пример 11. Взаимозависимые типы.

```
контекст
    тип анк; тип ам;        % предварительные описания
    тип мж = (м, ж),
    обланк = область [до 10000]     % анкеты
        структ (длим#до 30)
        (конст имя: стрлит (длим),
        датар: структ (
                конст год#до 2000,
                мес#до 12, день#до 31),
        пол#мж;
        перем супруг#анк, личнам#ам),
    облам = область [до 10000]     % автомашины
        структ (конст номер: стрлит (7),
                владелец#анк) :
    конст арханк#обланк,     % архив анкет
          архам#облам;       % архив данных об а/м
    тип анк = элем арханк,
        ам = элем архам;
    конст арханк = ген обланк (1000) ,
          архам = ген облам (1000)
начало
    перем муж#, жена#анк, маш#ам;
    муж:= ген анк (20) иниц
        (имя : стр8 "иванов иван иванович",
        датар: (год : 1946, мес : 4, день : 1), пол : м) ;
    жена := ген анк (22) иниц
        (имя : стр8 "иванова ольга петровна",
        датар: (год : 1954, мес : 5, день : 3),
        пол : ж, супруг : муж) ;
    маш := ген ам иниц
          (номер: стр8 "д8172мо", владелец: муж) ;
        муж.супруг := жена ;
        муж.личнам := маш;
    . . .
конец     % примера 11.
```

Пример 12. Модульный тип, доопределение модуля, включение в контекст.

```
контекст
% программа обработки двунаправленной цепочки термов.
% термы состоят из четырех элементов:
% 1) целочисленный ключ;
% 2) строка литер;
% 3) ссылка на предшествующий терм цепочки;
% 4) ссылка на следующий терм цепочки.
% встроенный пакет обработки цепочек обеспечивает создание
% термов, их уничтожение и упорядоченность по ключам.
  конст обрцепь :
    интерфейс        % описание интерфейса пакета
                     % обработки цепочек.
      тип звено;     % предварительное описание типа.
      конст          % предварительное описание интерфейсных
                     % процедур пакета.
      создцепь: функция ( )#звено,
      добвцепь: проц (перем нач#звено;
                      конст х#цел,
                           у# стрлит),
      найти: функция (конст нач#звено, х#цел)#звено,
      удалить: проц (перем нач#звено; конст х#цел)
    конинт,                % конец интерфейса
      обрцепь =            % доопределение пакета обработки
                           % цепочек термов
контекст     % скрытая часть пакета.
    тип цепь = область [до 1000]
      структ (перем пред#, след#звено; конст ключ#цел,
      стр# стрлит) );
    конст куча#цепь
начало       % доопределение интерфейсной части пакета.
    тип звено = элем куча;
    конст
    куча = ген цепь(1000),
    ник = пусто звено, % пустой элемент — эквивалент nil
    создцепь = функция( )#звено (ник),
    добвцепь = проц (перем нач#звено;
                     конст х#цел, у# стрлит)
      начало
        конст нов#звено = ген звено иниц (ключ : х, стр : у) ;
        перем элтек#звено := нач, элпред#звено := ник;
        % найти элемент, ключ которого превышает х
        до найден, ненайден
        цикл
          если элтек <=> ник то ненайден!
          инес элтек. ключ > х то найден!
```

```
        иначе элпред := элтек; элтек := элтек.след
        все
      повторить;
      % вставить новый перед найденным элементом
      нов.пред := элпред; нов.след := элтек;
      если не (элтек <=> ник) то элтек.пред := нов все;
      если не (элпред <=> ник) то элпред.след := нов
      иначе нач := нов все
    конец ,     % добвцепь
  найти = функция (конст нач#звено, х#цел)# звено
    начало
      перем элтек#звено := нач;
      до найден
      цикл
        если элтек <=> ник то найден!
        инес элтек.ключ = х то найден!
        иначе элтек := элтек.след все
      повторить ;
      элтек
    конец,          % найти
  удалить = проц (перем нач#звено; конст х#цел)
    начало
      перем элх#звено := найти (нач,х) ;
      если не (элх <=> ник) то % найден х
        если элх.пред <=> ник то      % удалить первое звено
          нач := элх.след:
          нач.пред := ник
        инес элх.след <=> ник то      % удалить последнее звено
          элх.пред.след := ник
        иначе
          элх.след.пред := элх.пред;
          элх.пред.след := элх.след
        все все
      конец
  конец; % доопределения пакета
    обрцепь    % включение в контекст и использование пакета
               % обработки цепочек.
начало
    перем цепь1#, цепь2#звено;
    цепь1 := создцепь ( );
    цепь2 := создцепь ( ) ;
    добвцепь (имя цепь1,5, стр8"авс");
    добвцепь (имя цепь2,6,стр8 "def")
    . . .
  конец    % примера 12.
```

Пример 13. Реализация примитивов работы со списками языка Лисп.

Списки в этом примере рассматриваются как объекты скрытого (абстрактного) типа. Тип и процедуры обработки списков предварительно описаны в интерфейсе, архивное имя которого — *//интлисп*. Реализация этого пакета находится в архиве под именем *//рлисп*.

Завершает пример программа, использующая этот пакет. Процедуры пакета реализованы таким образом, что в случаях, оставленных неопределенными в первоначальном описании Лиспа [107], возникает ситуация *неверныйоперанд*.

```
% файл //интлисп
интерфейс
   тип лисп;      % предварительное описание типа, определяю-
                  % щего структуру списка.
   конст
   car : функция (конст x#лисп)#лисп,
   cdr : функция (конст x#лисп)# лисп,
   cons : функция (конст x#, y#лисп)#лисп,
   atom : функция (конст x#лисп)#лог,
   eq : функция (конст x#, y#лисп)#лог,
   null : функция (конст x#лисп)#лог,
   equal : функция (конст x#, y#лисп)#лог
конинт
% файл //рлисп
реал //интлисп
контекст -
   конст вж = генвж( ) ;
   % объекты, создаваемые с помощью процедур генерации,
   % должны иметь такое же время жизни
   % как и описываемый модуль;
   % константа вж создана, чтобы указывать в генераторах
   % это время жизни.
   тип са = (сп, ат)      % тег элемента списка.
начало
   % далее следует доопределение интерфейсных процедур.
   тип лисп = выбтип са из
              ат: унчис,
              сп: структ (конст carx#, cdrx# лисп)
              всевыб;
   конст
   car = функция (конст x#лисп)# лисп
         (x#лисп (сп). carx),
```

```
   cdr = функция (конст x#лисп)# лисп
         (x#лисп (сп).cdrx),
   cons = функция (конст x#, y#лисп)# лисп
         (ген лисп (сп) [локал : вж] иниц (carx : x, cdrx : y) ),
   atom = функция (конст x#лисп)# лог
         (провтип (x, лисп (ат) ) ),
         % предописанный в эль-76 предикат. Его значение —
         % истина, если x является объектом указанного типа.
   eq    = функция (конст x#, y#лисп)# лог
           (x# лисп (ат) = y# лисп (ат) ),
   null  = функция (конст x#лисп)# лог
           если atom(x) то eq (x, "nil") иначе ложь все,
   equal = функция (конст x#, y#лисп)# лог
           если atom (x) то
              если atom (y) то eq (x, y)
              иначе ложь все
           инес не atom (y) то
              если equal(car(x), car (y))
              то equal (cdr (x), cdr (y) )
              иначе ложь все
           иначе ложь все
конец   % реализации примитивов лиспа.
```

Программа, использующая описанный пакет.

```
контекст
   конст a: //интлисп = (//плисп) ; a
начало
   перем список1#, список2#лисп;
   перем атом1, атом2#лисп;
   . . .
   список1 := cons ("язык", "эль-76") ;
   список2 := cons (список1, "описание") ;
   . . .
   атом1 := car (car (список2) ) ;
   атом2 := cdr (список1)
конец   % примера 13.
```

Пример 14. Определение интерфейса и тела модуля. Совместное использование статики и динамики типов.

В данном примере приводятся описания типа и тела модуля, обеспечивающего хранение информации любого типа в скрытом от пользователя буферном массиве. Запись информации в массив и ее считывание осуществляется с помощью интерфейсных процедур *положить* и *взять*. При переполнении буфера и при чтении из пустого

буфера возникают ситуации *переполнен* и *пустой*. Программа, описывающая интерфейс модуля, имеет вид:

```
интерфейс
   конст положить : проц (конст x),
      взять     : функция ( ),
      переполнен = ситуация ( ),
      пустой = ситуация ( )
конинт
```

Будем считать, что эта программа помещена в архив под именем *//буфер*. Тело модуля, реализующее обработку буфера по схеме «первый записанный — первый прочитанный», имеет вид:

```
реал //буфер
контекст     % скрытая часть
   конст м = 100, буф#свдин = ген свдин (м);
   перем укзап#цел := — 1,
      укчит#цел : = —1,
      к#цел := 0;
начало     % доопределение интерфейса
   конст
   положить = проц (конст x)
      если к + 1 <= м то     % можно положить
         к := к + 1;
         укзап := (укзап + 1) остат м;
         буф [укзап] := x
      иначе переполнен!
      все,
   взять = функция ( )
      если к > 0 то     % в буфере есть информация
         к := к — 1;
         укчит := (укчит + 1) остат м;
         буф [укчит]     % результат функции
      иначе пустой!
      все
конец
```

Буферный массив этого модуля состоит из переменных динамического типа. Этот модуль могут использовать программы, хранящие в буфере информацию различных (в общем случае любых) типов. Будем считать, что эта реализация помещена в архив под именем *//буфоч*.

Пример программы, хранящей в буферном массиве целочисленную информацию, может выглядеть следующим образом:

```
контекст
   конст обрбуф : //буфер = (//буфоч) ; обрбуф;
   перем х#, у#цел;
начало
   до * переполнен, * пустой
      (. . .
      положить(х);
      . . .
      у := взять( )#цел;
      . . .);
      . . .
конец
```

Приведем пример другой реализации того же интерфейса по схеме «последний записанный — первый считанный». Кроме того, введем параметр реализации — размер скрытого буфера.

```
реал //буфер (конст м#цел)
контекст     % скрытая часть
   конст буф#свдин = ген свдин (м);
   перем к#цел := —1
начало     % доопределение интерфейса
   конст
   положить = проц (конст х)
      если к + 1 < м то  % можно положить
         к := к + 1;
         буф [к] := х
      иначе переполнен!
      все,
   взять = функция ( )
      если к >= 0 то
         к := к — 1;
         буф [к + 1]
      иначе пустой!
      все
конец
```

Пусть архивное имя этой программы — *//буфстек*. Приведем пример ее использования:

```
контекст
   конст обрбуф : //буфер = (//буфстек, 1000);
   . . .
   % далее аналогично примеру с //буфоч
   . . .
конец
```

Пример 15. Организация взаимоисключающего доступа из параллельных процессов.

Если организован взаимоисключающий доступ к модулю, его можно использовать в параллельных процессах. Например, модуль обработки буфера (см. пример 14) можно использовать в параллельных процессах типа «писатель/читатель». Для этого вместо ситуаций *переполнен* и *пустой* надо описать в реализации локальный семафор, а в процедурах *положить* и *взять* запрограммировать синхронизацию по доступу к буферу. Интерфейс модуля изменится следующим образом:

```
интерфейс
   конст положить : проц (конст x),
        взять : функция ( )
конинт
```

Реализация модуля *//буфстек* с взаимоисключающим доступом может выглядеть следующим образом:

```
реал //буфер (конст м#цел)
контекст
   конст буф#свдин = ген свдин (м);
   перем к#цел := -1,
        блокирбуф := семоткр
начало
   конст
   положить = проц (конст x)
      до записано
      цикл    % блокировать доступ
         закрытьсем (имя блокирбуф);
         если к + 1 <= м то
            к := к + 1;
            буф [к] := x;
            открытьсем (имя блокирбуф) ;
            записано!    % разблокировать и выйти
         иначе
            открытьсем (имя блокирбуф)
            % надо разблокировать и ждать место
         все
      повторить,
   взять = функция ( )
      начало
        перем x;    % локал
        до прочитано
        цикл
          закрытьсем (имя блокирбуф) ;
          если к >= 0 то    % есть информация
```

```
            к := к — 1;
            х := буф [к];
            открытьсем (имя блокирбуф);
            прочитано! (х)
        иначе    % надо ждать информацию
            открытьсем (имя блокирбуф)
        все
      повторить
    конец
конец
```

Чтобы повысить эффективность выполнения этой программы, можно приостанавливать процесс, если нет информации для чтения или нет места для записи. Для этого надо ввести событийный семафор *семзакр* и воспользоваться примитивами *ждать* и *пропустить*.

Пример использования модуля в параллельных процессах:

```
начало
  конст обрбуф : //буфер = (//буфстек);
  процедура писатель = проц (. . .)
    начало
      . . .
      обрбуф. положить (. . .);
      . . .
    конец;
  процедура читатель = проц (. . .)
    начало
      . . .
      обрбуф. взять ( )
      . . .
    конец;
  . . .   % другие описания
  создпроцесс (1, писатель, . . .параметры...);
  создпроцесс (1, читатель, . . .параметры. . .);
  . . .
конец
```

Пример 16. Использование формального контекста и нескольких реализаций для одного интерфейса.

В этом разделе дан пример описания универсального алгоритма, применимого для решения задач из разных предметных областей. При этом используется возможность иметь несколько реализаций для одного интерфейса. В приведенном ниже примере описан рекурсивный алгоритм поиска решения некоторой задачи методом проб и ошибок с возвратами. Алгоритм сформулирован в терминах абстрактных понятий: «инициировать перебор вариантов», «следующий

вариант» и т. д. Каждому понятию соответствует процедура: *инициперебор*, *следвар* и т. д. Предполагается, что набор этих процедур представляет собой набор операций для анализа возможных вариантов решений задач в некоторой предметной области. Они объединяются в следующий интерфейс «анализа вариантов»:

```
интерфейс
   тип тпоз;    % предварительное описание типа позиции;
                % полное описание будет дано в доопре-
                % делении модуля
   конст
   иницперебор : функция (конст поз#тпоз, вж)# тпоз,
   следвар     : проц (конст поз#тпоз)
   приемлем    : функция (конст поз#тпоз)# лог,
   запшаг      : проц (конст поз#тпоз),
   решполное   : функция (конст поз#тпоз)# лог,
   ликвидшаг   : проц (конст поз#тпоз),
   печатьреш   : проц ( ) ,
   исхпоз# тпоз,
   исчерпвар = ситуация ( )
конинт
```

Предположим, что эта программа помещена в архив под именем *//анализвар*. Приведем программу поиска решения.

```
контекст (абстракция#//анализвар) % формальный контекст
реал
проц ( )    % программа-процедура работает в контексте набора
            % операций анализа вариантов.
начало
конст успех=ситуация ( ),  % сигнал о том, что полное
                           % решение найдено
   тупик = ситуация ( ); % сигнал о том, что процесс зашел в
                           % тупик и требуется возврат на
                           % несколько шагов
   процедура поиск = проц (конст поз#тпоз)
      % процедура рекурсивного спуска по дереву возможных
      % вариантов; поз — исходная вершина
      цикл % перебора возможных вариантов (вар) из вершины
           % поз
         конст вж = генвж ( ), вар#тпоз = иницперебор (поз, вж);
         до * исчерпвар      % ситуация возникает в следвар при
                             % исчерпании вариантов
         (следвар (вар) )
         при исчерпвар: тупик!
         всесит;
```

```
        % проверим приемлем ли вариант
        если приемлем (вар) то
          запшаг (вар) ;     % регистрируем продвижение к
                             % решению на 1 шаг
          если решполное (вар) то успех!
          иначе
            до * тупик
            (поиск (вар) )
            при тупик: ликвидшаг (вар)
            всесит
          все
        все
      повторить;     % перейти к следующему варианту
    до * успех, *тупик
    (поиск (исхпоз) )
    при успех: печатьреш ( ),
      тупик: % печать сообщения об отсутствии решения
    всесит
конец
```

Чтобы применять эту программу для решения различных задач, будем для каждой задачи описывать свою реализацию интерфейса *//анализвар*. Рассмотрим две задачи:

1) путешествие шахматного коня: построить алгоритм обхода конем шахматной доски размером $м \times м$ так, чтобы конь побывал в каждой клетке ровно один раз;

2) расстановка ферзей: расставить $м$ ферзей на доске размером $м \times м$ так, чтобы ни один из них не находился под ударом другого.

Задача коня.

```
реал //анализвар (конст м#до 10)     % размер доски
контекст
  % в этой реализации позиция идентифицируется пятеркой
  % чисел x, y, xx, yy, номвар.
  тип тпоз = структ (
    перем x#, y#,          % исходная позиция
          xx#, yy# до 9,   % текущая позиция
          номвар#до 8):    % номер варианта
  перем к#цел:= 0;         % номер хода
  тип т=массив [м,м] перем#цел;
  конст
  н#т = ген т,     % доска м*м
                   % массивы для вычисления хода коня
  a : массив [9] конст#цел = ( 1 : 2, 2 : 1, 3 : —1, 4 : —2,
                               5 : —2, 6 : —1, 7 : 1, 8 : 2),
```

```
   в : массив [9] конст#цел = ( 1 : 1, 2 : 2, 3 : 2, 4 : 1,
                                5 : —1, 6 : —2, 7 : —2, 8 : —1)
начало
   конст
   приемлем = функция (конст поз#тпоз)# лог
      (поз. хх = 0 и поз. уу = 0),
   решполное = функция (конст поз#тпоз)# лог
      (к = м**2),
   иницперебор = функция (конст поз#тпоз, вж)# тпоз
      начало
         конст нач#тпоз = ген тпоз [локал : вж] иниц (номвар:0);
         если не (поз <=> исхпоз) то      % не начальная позиция
            нач.х := поз.хх; нач.у := поз.уу
         иначе нач.х := 0; нач. у := 0 все;
         нач
      конец,
   следвар = проц (конст поз#тпоз)
      до найденвар
      цикл
         перем хх#, уу#цел;
         если поз.номвар = 8 то исчерпвар! все;
         поз.номвар := поз.номвар + 1;
         хх := поз.х + а [поз.номвар] ;
         уу := поз.у + в [поз.номвар] ;
         если (0 <= хх и хх <= м — 1) и
         (0 <= уу и уу <=м — 1) то
         поз.хх := хх; поз.уу := уу;
         найденвар!
         все
      повторить,
   запшаг = проц (конст поз#тпоз)
      (к := к + 1; н [поз.хх, поз.уу] := к),
   ликвидшаг = проц (конст поз#тпоз)
      (к := к — 1; н [поз.хх, поз.уу] := 0),
   печатьреш = проц ( )
      (. . .печать массива н в наглядном виде. . .),
   исхпоз = пусто тпоз
конец
```

Будем предполагать, что эта программа помещена в архив под именем *//конь*.

Расстановка ферзей.

```
реал //анализвар (конст м#до 8)
контекст
```

```
тип
  тпоз = структ (перем x#, y# цел);
    % в этой реализации структура типа тпоз идентифицирует
    % установку ферзя в поле ⟨x, y⟩.
  конст
  реш#свкор = ген свкор (м),  % вектор решения
                              % реш [x] положение ферзя на
                              % горизонтали x.
  a#свлог = ген свлог (м) иниц (: истина),
  в#свлог = ген свлог (2*м — 2) иниц (: истина),
  с#свлог = ген свлог (2*м — 2) иниц (: истина);
    % векторы a, в, с нужны для быстрой проверки
    % условия, что данное поле не бьется:
    % a [x] и в [y + x] и с [y — x + м — 1] = истина,
    % если поле ⟨x, y⟩ не бьется.
начало
  конст
  приемлем = функция (конст поз#тпоз)# лог
    (a [поз.x] и в [поз.y + поз.x] и с [поз.y — поз.x + м — 1]),
  решполное = функция (конст поз#тпоз)# лог
    (поз. y = м — 1),
  иницперебор = функция (конст поз#тпоз, вж)# тпоз,
    начало
      конст нач = ген тпоз [локал: вж] иниц (x : —1);
      если не (поз <=> исхпоз) то
        нач.y := поз.y + 1
      иначе нач.y := 0 все;
      нач
    конец,
  следвар = проц (конст поз#тпоз)
    если поз.x = м — 1 то исчерпвар!
    иначе поз.x := поз.x +1 все,
  запшаг = проц (конст поз#тпоз)
    (реш [поз.x] := поз.y;
    a [поз.x] := в [поз.y + позх] :=
    с [поз.y — поз.x + м —1] := ложь),
  ликвидшаг = проц (конст поз#тпоз)
    % действие, обратное производимому
    % процедурой запшаг
    (a [поз.x] := в [поз.y + поз.x] :=
    с [поз.y — поз.x + м — 1]:= истина),
  печатьреш = проц ( )
    (% печать вектора реш
    % в наглядном виде
    ),
```

исхпоз = **пусто** *тпоз*
конец

Будем считать, что эта программа помещена в архив под именем *//ферзи*.

Программа решения задачи коня на доске 10*10 выглядит следующим образом:

```
начало
   конст задачаконя: //анализвар = (//конь, 10);
      % генерация модуля типа //анализвар
      % с реализацией //конь
   ген проц ( ) иниц (//поиск, задачаконя) ( )
      % генерация и вызов экземпляра процедуры //поиск,
      % настроенной для решения задачи коня
конец
```

Программа решения задачи о ферзях на доске 8*8:

```
начало
   конст задачаферзей: //анализвар = (//ферзи, 8),
      % генерация модуля типа //анализвар
      % с реализацией //ферзи
   анализатор: проц ( ) = (//поиск, задачаферзей);
      % генерация экземпляра процедуры //поиск,
      % настроенной на решение задачи ферзей
   анализатор ( )
      % вызов этой процедуры для решения
      % задачи о ферзях
конец
```

Программы решения обеих задач можно слить в одну. Формально это означает, что в процессе ее выполнения будут создаваться два экземпляра модуля *//анализвар* с разными реализациями и два экземпляра процедуры *//поиск* (каждый настроен на соответствующий модуль).

ГЛАВА 3

СРЕДСТВА ВЗАИМОДЕЙСТВИЯ С ВНЕШНИМИ ОБЪЕКТАМИ

В этой главе описываются средства организации обмена и архивного хранения данных на внешних устройствах.

Конструкции, связанные с обменом, по своим изобразительным средствам разбиваются на две категории:

1. Конструкции, обозначающие генераторы объектов и основные функциональные операции обмена *). Эти конструкции имеют нестандартный синтаксис вызова процедуры (см. пп. 2.1, 7.1) и начинаются с идентификаторов *чит, зап, нов, уст, читпар, заппар, устпар, читатр, запатр, генфайл, генконтейнер, файл, контейнер, позп, бвв, тбуф.*

2. Конструкции, обозначающие дополнительные операции. Эти конструкции имеют стандартный синтаксис вызова процедуры, например *ликвидвнеш (объект)*. Параметры в общем случае могут иметь форму (выражения).

■ В описании семантики операций, кроме их логического смысла, приводится системная интерпретация, т. е. описываются действия, которые фактически производит внешнее устройство. В этой связи необходимо отметить, что в случае строго вводных устройств (чтение с перфокарт, чтение с перфоленты) и строго выводных устройств возможно промежуточное хранение информации на магнитных дисках или барабанах (виртуальные файлы). Для выводных файлов это означает, что информация записывается в виртуальный файл и лишь впоследствии целиком выводится на реальное внешнее устройство. Для вводных файлов это означает, что информация сначала целиком передается в виртуальный файл, а программа затем

*) Под термином «функциональные операции» здесь понимаются операции, реализованные в виде процедур операционной системы. В дальнейшем слово «функциональные» будет для краткости опускаться.

считывает из этого файла. Системная интерпретация, приведенная в семантике, не учитывает возможности промежуточного хранения, а описывает гипотетическую картину непосредственного общения программы с внешним устройством. ■

1. Объекты. Общие сведения

1.1. Компоненты внешних объектов. Основным представителем внешних объектов является файл. Файл состоит из двух основных компонент — заголовка файла и его содержимого — последовательности элементов файла. Доступ к содержимому осуществляется косвенно посредством заголовка файла. Кроме того, в заголовке файла хранятся его атрибуты, т. е. ряд физических и логических характеристик файла.

Содержимое файла может иметь некоторую логическую структуру. Например, это может быть последовательность строк текстовой информации или объектный код программы.

Под термином «внешние объекты» подразумеваются не только файлы в традиционном смысле, но и такие образования, как программы, справочники и контейнеры. Контейнер состоит из последовательности пронумерованных математических томов. Математическому тому поставлен в соответствие физический том. Томом контейнера на магнитных барабанах или дисках является область на барабане или пакет дисков. Контейнер на барабанах или дисках может состоять из одного или нескольких томов, причем в состав одного и того же контейнера могут входить и барабаны и диски или только барабаны или только диски. Ниже для контейнеров на барабанах и/или дисках употребляется сокращение «мд-контейнер». Томом контейнера на магнитной ленте (мл-контейнера) является бобина магнитной ленты. Мл-контейнер состоит из одного тома.

В последующем описании различаются следующие разновидности внешних объектов:

1. *Файл на магнитных барабанах* (*мб-файл*) *или магнитных дисках* (*мд-файл*). Компонентами файла являются заголовок, последовательность элементов и справочник внешних связей. Каждый файл в свою очередь является компонентой некоторого мд-контейнера.

2. *Мд-контейнер*. Компонентами мд-контейнера являются заголовок, мб- или мд-файлы, базовый справочник контейнера и другие справочники, расположенные в данном контейнере.

3. *Файл на магнитной ленте* (*мл-файл*). Компонентами файла являются заголовок и последовательность элементов. Каждый файл в свою очередь является компонентой некоторого мл-контейнера.

4. *Мл-контейнер*. Компонентами мл-контейнера являются заголовок и мл-файлы, расположенные в данном контейнере.

5. *Файлы на прочих носителях*, а именно на перфокартах (при вводе это *чпк-файлы*, а при выводе — *эпк-файлы*), на бумажном носителе (*ацпу-файлы*), на перфоленте (при вводе это *чпл-файлы*, а при выводе — *эпл-файлы*), на экране (*ацд-файлы*), на бумажном носителе пишущей машинки (*пм-файлы*). Компонентами этих файлов являются заголовок и последовательность элементов.

6. *Справочник.* Компонентами справочника являются элементы справочника. Элемент справочника может содержать ссылку на некоторый внешний объект или пустую ссылку. С элементом справочника связано символьное обозначение (имя) этого элемента. Элементы упорядочены по возрастанию величин имен (см. атрибут *имяэлспр*).

7. *Программа.* Программа является объектом, содержащимся в файле объектного кода.

1.2. Доступ к объектам. Представителем внешнего объекта в оперативной памяти является оперативный объект связи с внешним объектом. Такими объектами являются: объект связи с файлом, объект связи с контейнером и указатель внешнего объекта.

Связь с файлом. Для каждой разновидности обмена существует свой объект связи с файлом. Для синхронного непосредственного обмена (т. е. обмена «программный массив ↔ файл» синхронно с выполнением программы) используется заголовок открытого файла. Заголовок открытого файла представляет собой копию заголовка файла в оперативной памяти. Для параллельного непосредственного обмена (т. е. обмена «программный массив ↔ файл» параллельно с выполнением программы) используется блок ввода/вывода (БВВ). БВВ в свою очередь связан с заголовком открытого файла. Для однобуферного обмена (т. е. обмена «один рабочий буфер ↔ файл») используется позиционная переменная. Позиционная переменная в свою очередь связана с заголовком открытого файла. Для многобуферного обмена (т. е. обмена «несколько рабочих буферов ↔ файл») используется таблица буферов. Таблица буферов в свою очередь связана с заголовком открытого файла.

Чтобы начать обмен, задача должна открыть файл, т. е. создать в оперативной памяти соответствующий объект связи с файлом.

Связь с контейнером. В ряде операций над контейнером требуется, чтобы контейнер был в открытом состоянии. Доступ к открытому контейнеру осуществляется косвенно посредством заголовка открытого контейнера. Заголовок открытого контейнера представляет собой копию заголовка контейнера в оперативной памяти.

Указатель внешнего объекта — это объект связи, который содержит ссылку на некоторый внешний объект или элемент справочника. Будем говорить, что указатель установлен на данный объект или элемент. С помощью соответствующих операций указатель можно установить на другой внешний объект или элемент справоч-

ника. Причем если указатель устанавливается на файл или контейнер, то это не приводит к открытию последних.

Доступ к справочнику осуществляется с помощью указателя внешнего объекта, установленного на данный справочник. Доступ к элементу справочника осуществляется с помощью указателя внешнего объекта, установленного на данный элемент справочника.

Доступ к программе осуществляется с помощью объекта связи, являющегося представителем файла объектного кода, содержащего данную программу.

■ *Открытие программы.* Открыть можно не только файл или контейнер, но также и программу. В простейшем случае в результате открытия программы выдается процедура. Выполнение программы заключается в вызове этой процедуры.

В общем случае в тексте программы можно описать те действия, которые выполняются при ее открытии, и тот объект, который является результатом открытия. В частности, результатом открытия может быть некоторый внешний объект, или элемент справочника. Таким образом, программу также можно рассматривать как один из возможных путей доступа к внешнему объекту или элементу справочника. ■

Косвенный доступ к объекту. Для всех операций, работающих с внешним объектом как с единым целым, несущественно, какой оперативный объект используется для связи. Эти операции минуют все промежуточные звенья и доходят до целевого объекта. Кроме оперативного объекта, промежуточным звеном может являться:

1. Элемент справочника. В этом случае в качестве следующего звена рассматривается объект, на который ссылается элемент справочника.

■ 2. Программа. Если этот объект не является целевым, то он рассматривается как промежуточное звено. В этом случае программа открывается и результат открытия рассматривается в качестве следующего звена.

3. Файл объектного кода. Если целевой объект — не файл, то в качестве следующего звена рассматривается программа, содержащаяся в этом файле. ■

В дальнейшем там, где возможно, вместо названия объекта связи, обеспечивающего доступ к внешнему объекту, и вместо цепочки «указатель внешнего объекта → элемент справочника →. . . .→ целевой внешний объект» употребляется название целевого внешнего объекта. Например, ряд конструкций может оперировать как с открытым файлом, так и с закрытым. В этом случае вместо терминов «заголовок открытого файла», «позиционная переменная», «указатель, установленный на файл», «указатель, установленный на элемент справочника, ссылающийся на файл» и т. д. используется термин «файл». Кроме того, имея в виду возможный неявный

проход по промежуточным звеньям, будем говорить, что указатель внешнего объекта приводит к целевому объекту.

■ Эти же соглашения действуют и тогда, когда промежуточным звеном цепочки является файл объектного кода или программа. ■

Термин «тип внешнего устройства» обозначает тип устройства, на котором может располагаться внешний объект.

1.3. Создание внешних объектов. Внешний объект создается с помощью ⟨генератора внешнего объекта⟩, например:

*мбф*1 := *генфайл* (*стконт*)
мбфайл := *генфайл* (*стконт*)

здесь создаются два файла (с атрибутами, выбираемыми по умолчанию) в стандартном мд-контейнере;

*диск*1 := *генконтейнер* (*мдконт*)

здесь создается мд-контейнер (с атрибутами, выбираемыми по умолчанию);

*мдф*1 := *генфайл* (*диск*1)

здесь создается файл (с атрибутами, выбираемыми по умолчанию) в контейнере *диск*1;

*спр*1 := *генспр* (*стконт*)

здесь создается справочник в стандартном мд-контейнере.

В результате выполнения ⟨генератора внешнего объекта⟩, кроме внешнего объекта, создается объект связи с ним. Этот объект связи выдается в качестве результата генерации. В приведенных выше примерах объект связи присваивается переменным *мбф*1, *мбфайл*, *диск*1, *мдф*1, *спр*1.

1.4. Создание объектов связи. Оперативные объекты создаются с помощью ⟨генератора объекта связи⟩, например:

*мбф*1*нпс* := *файл* (*мбф*1)

здесь открывается файл *мбф*1, т. е. создается заголовок открытого файла. С помощью этого заголовка можно в дальнейшем осуществлять непосредственный обмен с файлом:

зап (*мбф*1*нпс*, *адреснач*1, *массив*1)

здесь в файл, начиная с адреса *адреснач*1, записывается массив *массив*1;

*мдф*1*буф* := *позп* (*мдф*1)

здесь, во-первых, открывается файл *мдф*1, во-вторых, создается позиционная переменная для буферизованного обмена с этим файлом. Позиционная переменная становится значением переменной *мдф*1*буф*.

С помощью этой позиционной переменной можно осуществлять однобуферный способ буферизованного обмена с данным файлом (см. § 10).

■ Ниже приведен пример, демонстрирующий создание указателя внешнего объекта. Этот указатель является подвижным, т. е. его можно устанавливать на различные внешние объекты:

*ув*1 := *генув*( ) ■

1.5. Внешний контекст. Элемент справочника содержит ссылку на внешний объект. Этот внешний объект является файлом, контейнером, справочником. Последний, в свою очередь, может содержать ссылки на внешние объекты. Совокупность, состоящая из справочника А и внешних объектов, прямо или косвенно (через посредство промежуточных справочников) достижимых из этого справочника, называется внешним контекстом. Справочник А называется корневым справочником этого внешнего контекста.

Внешний контекст пользователя — это внешний контекст, состоящий из объектов, к которым данный пользователь может иметь доступ. Корневой справочник внешнего контекста пользователя создается при регистрации этого пользователя в системе. Первоначально корневой справочник состоит из одного элемента. Этот элемент содержит ссылку на корневой справочник глобального внешнего контекста, доступного всем пользователям.

К корневому справочнику применимы все операции над справочниками (создание и уничтожение элементов, поиск элемента по его имени, запись в элемент ссылки на объект). Из программы пакетного задания корневой справочник пользователя доступен с помощью идентификатора *свойвк* (*свой внешний контекст*).

Внешний контекст программы. В тексте программы могут встречаться алфавитно-цифровые внешние имена, предназначенные для упоминания внешних объектов. Соответствующие объекты ищутся (по их именам) во внешнем контексте данной программы. Каждая программа может, вообще говоря, иметь свой собственный внешний контекст или контекст, который частично или полностью пересекается с контекстом других программ.

Внешний контекст задания. Пакетное задание — это программа, внешним контекстом которой является внешний контекст пользователя, составившего задание.

1.6. Операции над справочником. К числу основных операций над справочниками относятся создание элемента, уничтожение элемента и запись в элемент ссылки на объект или элемент другого справочника.

Создание элемента. Справочник можно расширять, создавая в нем новые элементы, например,

создэлспр (*спр*1, "*файл*3", *мбфайл*)

Здесь в справочнике *спр*1 создается элемент с обозначением "*файл*3" и в него заносится ссылка на файл *мбфайл*.

Эта же операция используется для создания элементов в корневом справочнике внешнего контекста программы (последний обозначается идентификатором *свойвк*). Например:

создэлспр (*свойвк*, "*файл*2", *мбфайл*)

К элементу, созданному в последнем примере, можно обращаться по внешнему имени *//файл*2 (см. ниже).

Запись ссылки. Содержимое элемента справочника можно изменять:

запспр (*//файл*2, *генфайл* (*стконт*))

Здесь в элемент "*файл*2" заносится ссылка на вновь созданный мбфайл, а старая ссылка уничтожается.

Уничтожение элемента. Справочник можно сокращать, уничтожая его элементы, например:

ликвидэлспр (*//файл*2)

Здесь элемент "*файл*2" ликвидируется. Объект, на который ссылается элемент, также ликвидируется, если на него нет других ссылок.

1.7. Именование внешних объектов. Простейшая форма конструкции (внешнее имя) предназначена для обозначения указателя, установленного на элемент какого-либо справочника из внешнего контекста программы. Например, *//файл*2 — это указатель, приводящий к объекту, ссылка на который содержится в элементе "*файл*2" корневого справочника внешнего контекста программы. Если объект, к которому приводит указатель — это файл, то его открытие можно записать следующим образом:

*ф*1 := *файл* (*//файл*2).

Другой пример, если в элементе "*пак*1" справочника содержится ссылка на мд-контейнер, то создание в нем файла можно записать следующим образом:

*мдф*1 := *генфайл* (*//пак*1)

Внешнее имя может быть многослоговым, например:

//спа//спв//х

Здесь элемент "*спа*" должен ссылаться на справочник, в котором элемент "*спв*", в свою очередь, ссылается на справочник, в котором элемент "*х*" ссылается на некоторый внешний объект. В целом данное многослоговое внешнее имя обозначает указатель, приводящий к этому последнему внешнему объекту.

Указатель, приводящий к объекту из внешнего контекста программы, можно пересылать без ограничений. Например, его можно передавать в качестве параметра в другую программу, возможно, имеющую другой внешний контекст. При этом контекстная информация не теряется.

1.8. Именование объектов на съемных носителях. Доступ к объекту на съемном носителе (диски, ленты, перфокарты и пр.) можно организовать двумя способами:

1. Можно ссылку на объект занести в элемент какого-либо справочника (в частности, — в элемент корневого справочника внешнего контекста программы), и затем использовать обозначение элемента, как указано выше.

2. Можно обращаться к объекту с помощью его собственной метки, точнее — алфавитно-цифрового имени, находящегося в метке файла или контейнера (см. атрибуты *имяфайла*, *имяконт*). Например, внешнее имя мл-контейнера с меткой "*арх*1" записывается следующим образом:

ксн//*арх*1

Здесь *ксн* — стандартный идентификатор, с которого начинаются внешние имена объектов на съемных носителях.

1.9. Ссылка на объект. Для реализации связи задачи с объектом и для связи одного объекта с другим система создает скрытые ссылки соответственно из задачи на объект и из объекта на объект. Как правило, эти ссылки в явном виде недоступны программе. Для выполнения операций над объектом программы используется объект связи. Этот объект является доступным программе эквивалентом скрытой ссылки. Например, при открытии файла:

*мбф*1 := *мбф*2 := *файл* (//*файл*2)

в документации о задаче появляется скрытая ссылка на открытый файл, а значением двух переменных *мбф*1 и *мбф*2 становится заголовок открытого файла. При этом регистрируется наличие одной скрытой ссылки (а не двух ссылок) из задачи на файл. Если присваивать этот заголовок другим переменным, передавать его как параметр и пр., то эти действия не увеличивают количество скрытых ссылок из задачи на файл.

Прикрепление и время жизни объекта. Прикрепление объекта В к другому объекту А состоит в том, что создается скрытая ссылка из А на В. Аналогично, прикрепление объекта В к задаче А состоит в том, что создается скрытая ссылка из задачи А на объект В. Вообще говоря, одновременно могут существовать несколько скрытых ссылок на данный объект из нескольких задач и/или объектов, или даже несколько скрытых ссылок из одной задачи и/или одного внешнего объекта.

Прикрепление происходит при создании нового объекта, при открытии внешнего объекта, при связывании одного объекта с другим.

Откреплeние объекта B от объекта A состоит в том, что ликвидируется скрытая ссылка из A на B. Аналогично, откреплeние объекта B от задачи A состоит в том, что ликвидируется ссылка из A на B. Если B прикреплен к задаче A с помощью нескольких ссылок, то количество этих ссылок уменьшается на 1.

После очередного открепления объекта может оказаться, что этот объект не прикреплен ни к одному объекту и ни к одной задаче. В этом случае от объекта открепляются все прикрепленные к нему, и объект ликвидируется. Таким образом, время жизни объекта определяется наличием скрытых ссылок на него из задачи и/или из других объектов.

Открепление объекта происходит при выполнении ряда операций над объектами связи, при выполнении операций явного открепления, по окончании выполнения задачи, к которой объект прикреплен. Все эти случаи открепления указаны в соответствующих разделах описания. Кроме того, как указано выше, откреплeние объекта B от A может происходить рекурсивно, как результат открепления объекта A от другого объекта или задачи.

1.10. Ликвидация объектов. Ликвидация оперативного объекта означает освобождение оперативной памяти, занимаемой этим объектом.

Ликвидация внешнего объекта означает в основном освобождение памяти, занимаемой этим объектом на внешнем носителе.

Ликвидация объекта производится либо неявно, если не осталось скрытых ссылок на объект (см. выше), либо явно с помощью процедуры *ликвидвнеш*. ■

2. Создание, открытие и ликвидация объектов

2.1. Генератор объекта. Описываемые ниже генераторы предназначены для создания внешних объектов и оперативных объектов связи.

Внешние объекты.

⟨генератор внешнего объекта⟩ ::=
⟨спецификация внешнего⟩ ({⟨выражение⟩,} ⟨выражение⟩
{, ⟨список установок атрибутов⟩})
⟨спецификация внешнего⟩ ::= *генфайл* | *генконтейнер* | *генспр*

⟨Спецификация внешнего⟩ определяет тип создаваемого объекта, а ⟨список установок атрибутов⟩ — его атрибуты. Значение второго ⟨выражения⟩ задает принадлежность создаваемого объекта. Этот параметр называется «основой генератора».

В случае спецификации *генконтейнер* основой генератора должен быть признак создания контейнера (см. п. 2.2).

Пример 1. Создание контейнера.

*дискпак*1 := *генконтейнер* (*мдконт*)

Помимо вновь создаваемых контейнеров, существует стандартный системный мд-контейнер, в котором пользователь имеет ограниченный администратором системы ресурс памяти, используемый для создания и хранения файлов (см. п. 2.2).

В случае спецификации *генфайл* основой генератора может быть:

1. Признак создания файла (см. п. 2.2).
2. Контейнер. В этом случае создается файл в данном контейнере.
3. Файл. В этом случае создается новый файл в том же контейнере, где находится исходный файл.

Пример 2. Создание файла в контейнере *дискпак*1 (см. пример 1) с установкой атрибутов.

*мдф*1: =*генфайл*
(*дискпак* 1, *длинлиста* : 1024, *максдлинблока*: 256)

В случае спецификации *генспр* создается новый справочник. Основой может быть:

1. Признак создания объекта в стандартном контейнере (см. п. 2.2). В этом случае создается справочник в стандартном контейнере.
2. Мд-контейнер. В этом случае справочник создается в указанном контейнере.

■ Используя первое ⟨выражение⟩, в конструкцию ⟨генератор внешнего объекта⟩ можно подать подвижный указатель. Этот указатель установится на вновь созданный объект. В противном случае объекты связи создаются неявно: при генерации контейнера создается локальный заголовок открытого контейнера, а при создании файла или справочника — локальный указатель, установленный на файл или справочник. Во всех случаях вновь созданный внешний объект прикрепляется к объекту связи, который выдается в качестве результата выполнения конструкции. Если объект создается в контейнере, то последний прикрепляется к этому объекту. ■

Объект связи.

⟨генератор объекта связи⟩ ::=
⟨спецификация объекта связи⟩
({⟨выражение⟩,} {⟨список установок атрибутов⟩})
⟨спецификация объекта связи⟩ ::=
файл | *позп* | *тбуф* | *бвв* | *контейнер* | *прогр* | *генув*

⟨Спецификация объекта связи⟩ определяет тип создаваемого объекта, а ⟨список установок атрибутов⟩ — его атрибуты. Значение

⟨выражения⟩ задает основу генератора. В случае спецификации *файл*, *ввв*, *позп*, *тбуф* основой генератора может быть файл (в этом случае устанавливается связь с данным файлом), либо признак создания файла или контейнер (в этом случае создается новый файл).

В зависимости от спецификации объекта производятся следующие действия:

файл — открывается файл. Это означает, что если в оперативной памяти еще нет копии заголовка данного файла, то она создается. В противном случае используется уже существующий заголовок открытого файла;

позп — файл открывается и для него создается позиционная переменная;

тбуф — создается таблица буферов;

ввв — файл открывается и для него создается блок ввода/вывода;

генув — создается подвижный указатель внешнего объекта. Первоначально указатель содержит пустую ссылку. Основа генератора в данном случае опускается.

При создании заголовка открытого файла (позиционной переменной, таблицы буферов, блока ввода/вывода) происходит прикрепление файла (заголовка открытого файла) к созданному объекту связи.

В генераторе блока ввода/вывода можно опускать основу. В этом случае создается блок ввода/вывода, не связанный с каким-либо файлом. К этому блоку можно в дальнейшем прикрепить заголовок открытого файла. Это делается путем модификации атрибута *прикрепфайл*.

Открытие контейнера. В случае спецификации *контейнер* основой генератора может быть контейнер или признак создания контейнера (в этом случае создается новый контейнер). Контейнер открывается; открытие происходит аналогично открытию файла, но с учетом того, что в данном случае объектом связи является заголовок открытого контейнера.

Результат. В перечисленных выше случаях значением конструкции является созданный объект связи.

Открытие программы. Если указана спецификация *прогр*, то основой генератора может быть:

1. Файл объектного кода программы (см. атрибут *типфайла*). В этом случае открывается содержащаяся в нем программа (см. § 12 гл. 2).

2. Текстовый файл (см. атрибут *типфайла*), содержащий текст программы на некотором языке программирования. Если справочник внешних связей (СВС) этого файла еще не содержит ссылку на файл объектного кода программы, то текст программы транслируется соответствующим транслятором (см. атрибут *имяяз*), ссылка на

полученный файл кода заносится в СВС исходного файла (см. атрибут *кодпрогр*) и программа открывается. В противном случае программа открывается по уже имеющейся ссылке.

Внешним контекстом программы, полученной в результате неявной трансляции, становится внешний контекст исходного текстового файла (см. атрибут *внешконт* и § 12). ■

Текущая позиция. Считается, что в процессе обмена со строго последовательными файлами (мл-, пк-, ацпу-, пл-, ацд-, пм-файлы) существует текущая позиция. Текущая позиция — это положение головки записи/считывания по отношению к началу файла. Аналогичное понятие существует и для мл-контейнера (подробнее см. § 6).

Непосредственно после того, как файл (контейнер) создан, текущая позиция установлена на начало файла (контейнера), т. е. в положение, предшествующее первому элементу файла (первому файлу контейнера). В процессе обмена текущая позиция перемещается по файлу (контейнеру), см. § 8.

Текущая позиция может быть установлена на конец файла (контейнера); такая позиция есть положение, следующее за последним элементом файла (последним файлом контейнера).

При открытии существующего или вновь созданного строго последовательного файла текущая позиция устанавливается на начало файла (относительно мл-файла и мл-контейнера см. дополнительно § 6).

Понятие «текущая позиция» используется и при описании буферизованного обмена с файлами, расположенными на устройствах любого типа. В процессе однобуферного обмена информация о текущей позиции хранится в позиционной переменной (см. § 10). Считается, что при создании позиционной переменной текущая позиция установлена на начало файла (исключение составляет случай мл-файла: вновь созданная позиционная переменная в начальный момент отображает фактическое текущее положение головки по отношению к текущей ленте файла). В последующем позиционная переменная отслеживает перемещение текущей позиции.

■ Оперативный объект, прикрепленный к модулю, может быть явно откреплен от него с помощью вызова процедуры *откреп*. Если явного открепления нет, то локальный объект неявно открепляется от модуля, к которому он прикреплен, при уничтожении модуля. В этом случае над объектом выполняются те же действия, что и при явном откреплении.

Глобальный оперативный объект может быть явно откреплен от задачи, к которой он прикреплен, с помощью вызова в этой задаче процедуры *откреп*.

Если к моменту окончания задачи оказывается, что остались глобальные ссылки из этой задачи на объект, то этот объект неявно

открепляется от задачи с уничтожением всех ссылок из задачи на него.

Оперативный объект можно использовать до тех пор, пока он прикреплен к модулю, задаче или другому объекту. ■

2.2. Признаки создания объектов. В пп. 1—4 ниже приведены идентификаторы, используемые как средство обозначения признаков генерации внешних объектов. Пп. 5 и 6 касаются тех случаев, когда значением основы генератора является контейнер.

1) *стконт, барабан.* Создается файл в стандартном контейнере. Если задан атрибут *типву*, то файл создается на носителе указанного типа, а иначе тип внешнего устройства определяется системой. В результате может быть создан либо мб-файл, либо мд-файл. Если создание нового файла приводит к превышению ресурсов пользователя на стандартном контейнере или к превышению ресурсов заданного типа, возникает ошибка «нет ресурса».

2) *мдконт.* Создается мд-контейнер и базовый справочник контейнера.

3) *млконт.* Тип внешнего устройства — магнитная лента. Создается мл-контейнер.

4) *зпк, ацпу, зпл.* Тип внешнего устройства — вывод на перфокарты, алфавитно-цифровое печатающее устройство, вывод на перфоленту. Создается зпк-, ацпу-, зпл-файл.

5) мд-контейнер. Создается файл внутри данного мд-контейнера. Атрибут *типву* файла и условие возникновения ошибки «нет ресурса» определяется, как в п. 1.

6) мл-контейнер. Создается мл-файл внутри данного контейнера. Новый файл создается на том месте, куда предварительно установлена текущая позиция (подробно см. § 6).

Если создание нового файла в контейнере требует превышения физического объема носителя, то возникает ошибка «нет ресурса».

■ *Системная интерпретация.* Интерпретация действий, производимых при создании внешнего объекта, состоит в следующем:

1) мб-, мд-файл. Если файл создается с помощью спецификации *генфайл*, то в контейнере, где создается файл, отводится место для его заголовка. Иначе создается только оперативный объект связи, а место для заголовка отводится в случае, если надо сохранить файл из-за наличия ссылок на этот файл из других внешних объектов.

2) мл-контейнер. Занимается свободное устройство и на него монтируется том контейнера.

3) мл-файл. Если перед созданием нового файла внутри контейнера какой-либо файл этого контейнера был открыт, то его необходимо закрыть. Новый файл создается в том месте, куда установлена текущая позиция (подробно см. § 6). На ленту записывается

начальная метка файла. Конечная метка файла записывается на ленту в случае, если после операции записи выполняется какая-либо операция над данным файлом, отличная от записи. Вновь созданный файл считается последним в контейнере.

4) зпк-файл. Занимается свободное устройство и выводится перфокарта с начальной меткой файла.

5) ацпу-файл. Занимается свободное устройство и выводится начальная метка ацпу-файл.

6) зпл-файл. Занимается свободное устройство. Пл-файлы не имеют начальных и конечных меток. ■

2.3. Операции открепления и ликвидации объектов.

1) *ликвидвнеш* (*объект*). Параметр *объект* — это объект связи. Если параметр отличен от указателя, установленного на элемент справочника, то операция осуществляет ликвидацию внешнего объекта, а иначе ликвидируется внешний объект, ссылка на который содержится в элементе справочника. Объект связи, не являющийся подвижным указателем, также ликвидируется; подвижный указатель сохраняется. В него заносится пустая ссылка, кроме случая, когда указатель установлен на элемент справочника. Операция не выдает значение.

■ При ликвидации внешнего объекта от него открепляются все прикрепленные к нему внешние объекты (что в свою очередь может привести к их уничтожению). Последующая попытка обращения к ликвидированному объекту со стороны данной задачи или других задач приводит к возникновению ошибки «нет внешнего объекта».

Системная интерпретация уничтожения внешних объектов:

— мб-, мд-файл. Освобождаются ресурсы, занимаемые файлом в содержащем его контейнере. Контейнер открепляется от файла;

— мл-файл. Если ликвидируемый файл является первым файлом содержащего его контейнера, то после начальной метки контейнера записывается конечная метка контейнера. В противном случае эта метка записывается после конечной метки файла, который непосредственно предшествует ликвидируемому. В обоих случаях это означает, что ликвидируется ссылка (если она есть) из контейнера на ликвидируемый файл и, кроме того, ликвидируются файлы контейнера, которые находятся после ликвидируемого (если такие есть). Затем контейнер открепляется от файла. Лента остается в позиции конечной метки контейнера и устройство не освобождается. ■

2) *ликвидэлспр* (*уэс*). Параметром *уэс* является указатель, установленный на элемент справочника. Операция ликвидирует этот элемент. Как следствие, возможна ликвидация объекта, ссылка на который содержится в элементе справочника. Это происходит при условии, что данная ссылка — единственная ссылка на объект,

3) *откреп (объект)*. Параметр *объект* — это объект связи с файлом или контейнером. Операция осуществляет закрытие файла или контейнера.

■ Закрытие состоит в том, что объект связи открепляется от соответствующего модуля или от задачи — в зависимости от того, к чему он прикреплен.

Если открепление объекта привело к тому, что не осталось ссылок на данный объект, то:

— данный объект уничтожается;

— от данного объекта открепляются все прикрепленные к нему объекты (внешние или оперативные), что в свою очередь может привести к уничтожению последних.

Операция не выдает значение.

Если в результате открепления внешнего объекта ликвидированы не все ссылки из других объектов связи и/или из задач на него, то фактически с внешним объектом никаких действий не производится.

Если же ликвидированы все подобные ссылки, но остались ссылки из внешнего контекста, то с внешним объектом производятся следующие действия:

— мл-файл. Контейнер открепляется от файла. В зависимости от атрибута *направление* лента перематывается на конец или на начало файла;

— зпк-файл. Выводится перфокарта, соответствующая конечной метке пк-файла, и устройство освобождается;

— ацпу-файл. Выводится конечная метка ацпу-файла и устройство освобождается;

— чпк-, зпл-, чпл-, ацд-, файл. Устройство освобождается. ■

3. Атрибуты объектов и доступ к атрибутам

Атрибуты создаваемого объекта (внешнего или объекта связи) определяются следующим образом. При создании внешнего объекта значение атрибута *типву* (*тип внешнего устройства*) задается процедурой, осуществляющей создание. Значения остальных атрибутов определяются исходя из ⟨списка установок атрибутов⟩ (см. § 18 гл. 2) генератора. Атрибуты, не указанные в списке, полагаются равными значениям, выбираемым по умолчанию.

Семантика атрибутов описана в гл. 4. Для каждого атрибута указано, является ли он атрибутом внешнего объекта или атрибутом открытия (см. ниже). В этой же главе приведены идентификаторы атрибутов. Каждый такой идентификатор описан в стандартном контексте программы как идентификатор константы. Значениями этих констант являются номера соответствующих атрибутов.

3.1. Принадлежность атрибутов. Различаются атрибуты внешнего объекта и атрибуты открытия (или атрибуты объекта связи).

Атрибуты внешнего объекта имеют следующие особенности:

1. Значение атрибута хранится в заголовке, находящемся на носителе вместе с самим внешним объектом.

2. Если атрибут не указан в ⟨генераторе внешнего объекта⟩, то его значение выбирается по умолчанию.

3. Если в ⟨генераторе объекта связи⟩ указан уже установленный атрибут внешнего объекта, то осуществляется контроль соответствия. При несоответствии возникает «ошибка атрибута».

Атрибуты открытия характеризуют текущее открытие внешнего объекта и имеют следующие особенности:

1. Значение атрибута хранится в оперативном объекте связи. В генераторе внешнего объекта можно указать умолчание для некоторых атрибутов открытия.

2. Если атрибут не указан в генераторе оперативного объекта, то он полагается равным значению, находящемуся в заголовке соответствующего внешнего объекта. Если значение, выбираемое по умолчанию, не было задано явно, то выбирается стандартное значение.

3. Используя указатель, установленный на внешний объект, можно модифицировать находящееся в заголовке умолчание для атрибута оперативного объекта.

3.2. Доступ к атрибутам. В этом пункте описаны особенности применения конструкций ⟨модификация атрибутов⟩ и ⟨запрос атрибутов⟩ (см. § 18 гл. 2) в случае объектов связи и внешних объектов.

Модификация атрибутов. Ряд атрибутов объектов связи и внешних объектов имеет одинаковые номера; соответствующие идентификаторы констант также совпадают. Чтобы избежать двусмысленности, вводится следующее соглашение о порядке установления принадлежности атрибута. Пусть первым параметром конструкции является объект связи X и требуется модифицировать атрибут A. Сначала проверяется, есть ли у объектов такого типа, как объект X, атрибут A. Если есть, то обрабатывается атрибут A объекта X. В противном случае считается, что A — это атрибут внешнего объекта.

В частности, если требуется модифицировать умолчание для атрибутов открытия или такой атрибут внешнего объекта, для которого имеется совпадающий по номеру атрибут объекта связи, то необходимо воспользоваться указателем, установленным на нужный внешний объект, передав его в качестве первого параметра в конструкцию ⟨модификация атрибутов⟩.

■ Уточнение принадлежности атрибута. Списку установок атрибутов может предшествовать ⟨уточнение⟩. Значение ⟨уточнения⟩ имеет следующий смысл:

— если объект является файлом, а уточнение — целым, то изменяются атрибуты листа памяти файла (для мб- и мд-файлов), причем целое определяет математический номер листа, атрибуты которого изменяются;

— если объект является мд-контейнером, а уточнение — целым, то изменяются атрибуты тома рассматриваемого контейнера, причем целое определяет математический номер тома, атрибуты которого изменяются;

— если объект является файлом, а уточнение — буфером, связанным с файлом, то изменяются атрибуты этого буфера. Уточнение можно задать с помощью номера блока. В этом случае изменяются атрибуты буфера данного блока. Такой метод используется только при многобуферном способе буферизованного обмена. При однобуферном способе атрибуты текущего буфера изменяются через позиционную переменную без использования ⟨уточнения⟩.

Пример. Распределение (явно заданное) физического листа для математического листа с номером, равным значению *теклист*. Лист отводится на томе класса 1.

запатр (*мдф*1, *теклист*, *класслиста* : 1, *распределен* : **истина**) ■

При модификации строковых атрибутов (*имяфайла*, *имяконт*, *сообщение*) значение второго ⟨выражения⟩ в ⟨списке установок атрибутов⟩ — это набор или литерная строка. В обоих случаях заданные таким образом литеры становятся содержимым атрибута.

При модификации содержимого области пользователя в заголовке файла (атрибут *областьзаг*) значение второго ⟨выражения⟩ может являться вектором элементов формата ф64. В этом случае значения элементов вектора становятся значениями элементов области пользователя. Если значением второго ⟨выражения⟩ является целое, то оно задает новый размер области заголовка.

Запрос атрибутов. Принадлежность запрашиваемого атрибута устанавливается так же, как и при модификации атрибута.

При запросе строковых атрибутов значением ⟨уточнения⟩ должен являться литерный вектор. Содержимое атрибута переписывается в этот вектор, и в качестве результата выдается подвектор, содержащий столько начальных элементов исходного вектора, какова длина строки, содержащейся в атрибуте. Если заданный вектор короче строки, то возникает ситуация *ситошатр*.

При запросе содержимого области пользователя в заголовке файла может быть указано ⟨уточнение⟩. Значением ⟨уточнения⟩ должен быть вектор переменных формата ф64. В ре-

вультате выполнения запроса значениями элементов вектора становятся значения элементов области пользователя. Если ⟨уточнение⟩ не указано, то результатом запроса является размер области пользователя.

Пример 1.

читатр (*ацпуф*, *страница*) % номер текущей страницы файла
% *ацпуф*

Пример 2.

читатр (*мдф*1, *теклист*, *распределен*)
% выдается **истина**, если математический лист с номером
% равным значению *теклист* уже распределен.

4. Указатель внешнего объекта

Различаются две разновидности указателей внешнего объекта:

1. Постоянный указатель. Такой указатель создается конструкциями ⟨внешнее имя⟩, ⟨изображение файла⟩ и др. Существует ряд стандартных указателей такой разновидности (см. ниже).

■ Постоянный указатель содержит ссылку в форме плана поиска (см. п. 4.1). Постоянный указатель нельзя изменять с помощью операций установки указателя.

2. Подвижный указатель. Такой указатель создается генератором со спецификацией *генув*. С помощью соответствующих операций подвижный указатель может быть установлен на тот или иной внешний объект (файл, контейнер, справочник), на элемент некоторого справочника, или может содержать пустую ссылку. Ссылка в подвижном указателе содержится в форме физической ссылки (см п. 4.1). Установка подвижного указателя на закрытый файл не приводит к открытию файла. Установка указателя на закрытый контейнер приводит к неявному открытию последнего. ■

Считается, что указатель, установленный на мд-контейнер, установлен также на базовый справочник этого контейнера.

Стандартные постоянные указатели:

1) *свойвк* — идентификатор константы, описанной по умолчанию в собственном контексте каждой программы (см. § 12, гл. 2). Значением константы *свойвк* является постоянный указатель, приводящий к корневому справочнику внешнего контекста данной программы.

2) *глобарх* — идентификатор константы, описанной в стандартном контексте. Значением этой константы является постоянный указатель, установленный на корневой справочник глобального внешнего контекста, доступного всем пользователям.

3) *ксн*, *нпо* — идентификаторы констант, описанных в стандартном контексте. Значением константы *ксн* является постоянный

указатель, приводящий к корневому справочнику контекста съемных носителей (см. п. 5.1). Именование объектов контекста съемных носителей описано в п. 5.4.

Значением константы *нпо* является постоянный указатель, приводящий к внешнему объекту. Стратегия постановки нужного объекта на устройство определяется оператором системы (см. атрибут имяфайла).

4) *пустовнеш* — идентификатор константы, описанной в стандартном контексте. Значением этой константы является постоянный указатель, содержащий пустую ссылку.

4.1. Форма ссылки на внешний объект. Ссылка, содержащаяся в указателе внешнего объекта, может иметь одну из двух форм:

1. Физическая ссылка, представляющая собой физический адрес объекта по внешнему полю.

2. План поиска, представляющий собой пару, состоящую из ссылки на справочник, с которого начинается поиск, и совокупности параметров поиска. Последняя может представлять собой последовательность алфавитно-цифровых слогов для поиска в контексте, целое число, являющееся номером элемента справочника внешних связей файла и пр. Поиск выполняется не в момент создания плана поиска, а при открытии указуемого объекта, при создании в нем новых объектов (если поиск приводит к контейнеру) и пр. В связи с этим:

— план поиска может, в частности, приводить к такому объекту или элементу справочника, который еще не существует;

— поиск по одному и тому же плану может в разное время приводить к различным объектам.

4.2. Операции над указателем. В описании операций используются следующие обозначения для параметров: *ув* — подвижный указатель; *увпф* — подвижный или постоянный указатель; *во* — объект связи.

Если указатель *ув* установлен на некоторый объект X или элемент справочника X, то это означает, что X прикреплен к *ув*. В результате установки *ув* на другой объект Z происходит открепление X от *ув* и прикрепление Z к *ув*.

1) *обкт* (*ув*, *во*). Пусть *во* — указатель (постоянный или подвижный), установленный на элемент справочника. В этом случае операция устанавливает *ув* на тот объект, ссылка на который находится в исходном элементе справочника. В остальных случаях *ув* устанавливается на внешний объект, непосредственно связанный с *во* (без прохода по промежуточным звеньям). Если *во* является постоянным указателем, содержащим план поиска, то операция осуществляет поиск.

2) *обкт* (*ув*). В этой операции указатель *ув* должен быть установлен на некоторый внешний объект или элемент справочника.

При этом условии *обкт* (*ув*) эквивалентно *обкт* (*ув*, *ув*), т. е. если *ув* первоначально установлен на элемент справочника, то в результате операции он устанавливается на объект, ссылка на который содержится в исходном элементе справочника. В остальных случаях операция дает тождественный результат.

3) *копирув* (*ув*, *во*). Эта операция отличается от *обкт* тем, что она не проходит элемент справочника: *ув* всегда устанавливается на внешний объект или элемент справочника, непосредственно связанный с *во*. ■

5. Работа с архивом

В этом параграфе излагаются общие сведения, касающиеся средств работы с архивом, описываются операции над справочниками и конструкция ⟨внешнее имя⟩.

5.1. Общие сведения. Различаются справочники двух разновидностей:

1. Архивный справочник. Это справочник в мд-контейнере, создаваемый с помощью ⟨генератора внешнего объекта⟩. С элементом справочника связано алфавитно-цифровое обозначение или имя. Элементы справочника упорядочены по именам (см. атрибут *имяэлспр*).

■ 2. Справочник внешних связей файла (СВС). Такой справочник находится в заголовке файла и создается вместе с файлом. Элементы справочника пронумерованы от 0 до *n*—1, где *n* — значение атрибута *длинсвс* файла. Объект, обеспечивающий доступ к файлу, одновременно обеспечивает доступ к СВС этого файла. ■

Базовый справочник контейнера. При создании мд-контейнера в нем автоматически создается архивный базовый справочник данного контейнера. Объект, обеспечивающий доступ к контейнеру, одновременно обеспечивает доступ к базовому справочнику. Используя операции над справочником, можно создавать в базовом справочнике элементы, записывать в них ссылки на объекты и пр.

Некоторые из описанных ниже операций определены над мл-контейнером. В этих случаях мл-контейнер рассматривается как архивный справочник, содержащий элементы, имена которых совпадают с именами файлов этого контейнера. К таким операциям относятся операции поиска по справочнику (*элспр*) и операции позиционирования справочника.

Простые и косвенные ссылки. Различаются две разновидности ссылок из элемента справочника: простые и косвенные. Разновидность ссылки не играет роли при «поиске по справочнику», но играет роль при «поиске по контексту».

Поиск по справочнику. Поиск элемента с именем *s* в архивном справочнике заключается в том, что в данном справочнике ищется элемент, имя которого совпадает со строкой *s*.

Поиск по контексту. Поиск элемента с именем *s* по внешнему контексту, корневым справочником которого является архивный справочник СПР, происходит следующим образом. Сначала производится поиск элемента с именем *s* в справочнике СПР. Если таковой не найден, то среди элементов СПР, содержащих косвенные ссылки, рассматривается первый (в смысле упорядоченности элементов по именам). Этот элемент должен в свою очередь ссылаться на некоторый архивный справочник СПР1. Алгоритм поиска повторяется в контексте с корневым справочником СПР1. Если в этом контексте имя не найдено, то рассматривается следующий элемент исходного справочника, содержащий косвенную ссылку, и т. д. Глубина рекурсии в алгоритме поиска не должна превышать 10. В противном случае возникает ошибка «нет в архиве».

■ *Форма ссылки.* Ссылка из элемента справочника (простая или косвенная) имеет форму физической ссылки (см. п. 4.1), которой могут быть представлены только:

— взаимные ссылки между объектами, содержащимися в одном и том же мд-контейнере (каталогизированном или некаталогизированном);

— ссылка из архивного справочника на контейнер (каталогизированный или некаталогизированный);

— взаимные ссылки между объектами, содержащимися в различных каталогизированных мд-контейнерах. ■

Справочник пользователя. Для каждого пользователя при его регистрации в системе создается корневой справочник внешнего контекста данного пользователя. Пользуясь операциями со справочниками, пользователь может создавать элементы в этом справочнике, заносить в них ссылки на вновь созданные внешние объекты и т. д., т. е. использовать этот справочник в качестве базы для создания личного архива.

Первоначально корневой справочник состоит из одного элемента, имеющего обозначение "*глобарх*". Этот элемент содержит косвенную ссылку на глобальный внешний контекст, доступный всем пользователям.

Обращение к корневому справочнику происходит следующим образом. Если программа является программой пакетного задания, то внешним контекстом программы является контекст пользователя, сформировавшего задание (пользователь идентифицируется «картой

пользователя», см. § 15). Следовательно, используя в задании идентификатор константы *свойвк*, можно получить указатель, приводящий к корневому справочнику внешнего контекста пользователя.

■ *Каталогизация.* Значение атрибута *катлг* предписывает каталогизировать внешний объект (мд-, мл-контейнер). При каталогизации метка объекта (см. атрибуты *имяконт* и *имяфайла*) и другие сведения о нем помещаются в каталог системы. ■

Съемные объекты. Именование объектов на съемных носителях (мд- и мл-контейнеры, мл-, чпк- и чпл-файлы) трактуется следующим образом. Предполагается, что существует глобально известный контекст съемных носителей. У этого контекста есть гипотетический корневой справочник. Значением константы *ксн*, описанной глобально для всех программ, является постоянный указатель, установленный на этот справочник. Поиск элемента с именем *s* состоит в том, что оператору выдается запрос на установку съемного носителя, содержащего объект с меткой *s* (см. атрибуты *имяфайла*, *имяконт*).

5.2. Операции над справочником. В описании операций используются следующие обозначения для параметров: *ув* — подвижный указатель, *во* — объект связи, *спр* — архивный справочник, контейнер или файл. В последних двух случаях операция выполняется соответственно над базовым справочником контейнера или над СВС файла; *спрарх* — архивный справочник или контейнер; *им* — параметр, который может быть (а) набором, возможно, неполным, или литерной строкой (в этом случае параметр задает обозначение элемента архивного справочника) или (б) целым числом (в этом случае параметр задает номер элемента СВС); *узс* — указатель (подвижный или постоянный), приводящий к элементу справочника.

■ Если элемент справочника содержит ссылку на объект X, то это означает, что X прикреплен к данному справочнику. В результате записи в данный элемент справочника новой ссылки происходит откребление X и прикрепление объекта, ссылка на который записывается в элемент справочника.

1) *элспр* (*ув*, *спр*, *им*), *элвк* (*ув*, *спрарх*, *им*). Операция *элспр* осуществляет поиск элемента, с обозначением *им*, в справочнике *спр*. Операция *элвк* осуществляет поиск элемента, с обозначением *им*, в контексте с корневым справочником *спрарх*.

Результатом операции является указатель *ув*, установленный на найденный элемент справочника. Если же элемент не найден, то возникает ошибка «нет в архиве».

2) *элспр* (*ув*, *им*), *элвк* (*ув*, *им*). Эти операции отличаются от приведенных выше тем, что указатель *ув* первоначально должен быть установлен на справочник, в котором производится поиск. Тогда:

элспр (*ув*, *им*) эквивалентно *элспр* (*ув*, *ув*, *им*)

элвк (*ув*, *им*) эквивалентно *элвк* (*ув*, *ув*, *им*) ■

3) *создэлспр* (*спр*, *им*). Операция создает в справочнике *спр* новый элемент. Обозначение элемента задается параметром *им*. В созданный элемент справочника заносится пустая ссылка.

Результатом операции является значение параметра *спр*.

4) *создэлспр* (*спр*, *им*, *во*), *создэлвк* (*спрарх*, *им*, *во*). Данные операции отличаются от приведенных выше тем, что созданный элемент инициализируется, т. е. в него заносится ссылка на внешний объект, а также атрибуты, ограничивающие права доступа. Для этого выполняется операция *обкт* (*ув*, *во*), где *ув* — некоторый временный указатель. Затем ссылка и атрибуты защиты копируются из указателя *ув* в элемент справочника.

В элемент справочника нельзя заносить ссылку на статический файл или статический справочник. Операция *создэлспр* заносит простую ссылку, а *создэлвк* — косвенную.

Результатом является значение первого параметра.

5) *запспр* (*уэс*, *во*), *запвк* (*уэс*, *во*). В элемент, на который установлен *уэс*, помещается ссылка на внешний объект, а также атрибуты *во*, ограничивающие права доступа. Для этого параметр *во* обрабатывается, как описано в п. 4. Операция *запспр* заносит простую ссылку, а операция *запвк* — косвенную.

Результатом является значение параметра *уэс*.

6) *элспр* (*пув*, *спр*, *мсим*). Операция создает программный указатель внешнего объекта. Параметры имеют следующие значения: *пув* — вектор из двух элементов формата ф64; *мсим* — литерная строка или набор. Эта строка (или набор) содержит последовательность алфавитно-цифровых слогов, подчиняющуюся синтаксису (многословного имени) (см. п. 5.4).

Операция записывает в первый элемент вектора значение второго параметра, а во второй — значение третьего параметра. Эта пара образует программный указатель, приводящий к некоторому внешнему объекту, находящемуся в контексте с корневым справочником *спр*. В качестве результата выдается постоянный указатель внешнего объекта, сформированный из вектора *пув*.

5.3. Операции позиционирования справочника. В операциях позиционирования справочника используется подвижный указатель, установленный в некоторую позицию справочника (УПС). УПС, так же как и указатель, установленный на элемент справочника, идентифицирует некоторый элемент. Различие их состоит в том, что в первом содержится признак доступа ко всему справочнику, и такой указатель можно с помощью приведенных ниже операций перемещать с одного элемента справочника на другой элемент того же справочника. Во втором случае переместить указатель с одного элемента на другой можно только с помощью операции *элспр*, т. е. в программе должен быть доступен не только какой-либо элемент справочника, но и весь справочник.

При выполнении операций *начспр* или *конспр* (установить на начало справочника, установить на конец справочника) в указатель заносится признак доступа. Операции *следэлспр* и *предэлспр* воспринимают УПС как указатель исходной позиции и устанавливают его в соседнюю позицию. Операция *текэлспр* аннулирует признак доступа ко всему справочнику. В операциях, отличных от операций позиционирования, *упс* эквивалентен указателю, установленному на элемент справочника.

В описании операций используются следующие обозначения: *ув*, *спр* — см. операции над справочником; *упс* — указатель позиции в справочнике.

1) *начспр* (*ув*, *спр*), *конспр* (*ув*, *спр*). Операция *начспр* устанавливает указатель *ув* на начало справочника, а операция *конспр* — на конец справочника.

Результатом операции является указатель *ув*, в котором установлен признак доступа.

2) *начспр* (*ув*), *конспр* (*ув*). Указатель *ув* должен быть установлен на справочник. Тогда:

начспр (*ув*) эквивалентно *начспр* (*ув*, *ув*)

конспр (*ув*) эквивалентно *конспр* (*ув*, *ув*).

3) *следэлспр* (*ув*, *упс*). В процессе выполнения операции различаются следующие случаи:

— *упс* установлен на начало справочника. В этом случае *ув* устанавливается на первый элемент справочника;

— *упс* установлен на некоторый элемент справочника. В этом случае *ув* устанавливается на следующий элемент;

— *упс* установлен на последний элемент или на конец справочника. В этом случае возникает ошибка «нет в архиве». *ув* устанавливается на конец справочника.

Результатом является значение параметра *ув*.

4) *предэлспр* (*ув*, *упс*). Семантика этой операции аналогична операции *следэлспр* с учетом замены направления перемещения на противоположное.

5) *текэлспр* (*ув*, *упс*). *ув* устанавливается на тот же элемент, что и *упс*, но в *ув* не ставится признак доступа ко всему справочнику. *упс* в этой операции не может быть установлен на начало справочника или на конец справочника. Результатом является значение параметра *ув*.

6) *начспр* (*ув*), *конспр* (*ув*), *следэлспр* (*упс*), *предэлспр* (*упс*), *текэлспр* (*упс*). Указатель *ув* первоначально должен быть установлен на справочник. Тогда:

начспр (*ув*) эквивалентно *начспр* (*ув*, *ув*)

конспр (*ув*) эквивалентно *конспр* (*ув*, *ув*)

следэлспр (*упс*) эквивалентно *следэлспр* (*упс*, *упс*)

предэлспр (*упс*) эквивалентно *предэлспр* (*упс, упс*)
текэлспр (*упс*) эквивалентно *текэлспр* (*упс, упс*) ■

5.4. Внешнее имя. ⟨Внешнее имя⟩ используется для обозначения постоянного указателя внешнего объекта.

⟨внешнее имя⟩ ::= {⟨идентификатор⟩} ⟨план поиска⟩
⟨план поиска⟩ ::= ⟨многослоговое имя⟩ | //[⟨выражение⟩]
⟨многослоговое имя⟩ ::= ⟨слог⟩ . . .
⟨слог⟩ ::= //⟨буква или цифра⟩. . . | //⟨стандартный слог⟩
⟨стандартный слог⟩ ::= *.обкт* | *.текст* | *.код* | *.прогр*

⟨Внешнее имя⟩ в виде ⟨многослогового имени⟩, не содержащего ⟨стандартных слогов⟩, обозначает указатель, установленный на элемент некоторого справочника. Последний находится во внешнем контексте программы.

Упомянутый элемент справочника можно найти следующим образом. Пусть ⟨внешнее имя⟩ имеет вид

$//s_1//s_2//. . .//s_n$

где $s_к$ — *к*-й слог, являющийся алфавитно-цифровым именем. Сначала во внешнем контексте ищется (см. п. 5.1) такой элемент справочника, обозначение которого совпадает с первым слогом (при неудовлетворительном результате поиска возникает ошибка «нет в архиве»). Если во ⟨внешнем имени⟩ есть второй слог, то найденный элемент справочника должен в свою очередь ссылаться на справочник (возможно, через посредство промежуточных звеньев). Этот справочник рассматривается как корневой, и процесс поиска повторяется уже со вторым слогом. Поиск по третьему и последующим слогам выполняется аналогично. В результате в некотором справочнике будет найден элемент, имя которого совпадает с последним слогом. Это и есть искомый элемент. Смысл ⟨стандартных слогов⟩ описан ниже.

Заметим, что описанный здесь алгоритм выполняется при непосредственном взаимодействии с объектом, например, в момент открытия. При выполнении конструкции ⟨внешнее имя⟩ поиск не происходит, а лишь создается указатель, содержащий план поиска.

■ Ссылка на внешний контекст программы находится в справочнике внешних связей файла объектного кода программы. Эту ссылку заносит транслятор после генерации файла или пользователь. Номер элемента, в котором должна содержаться ссылка, определяется во время трансляции и заносится в атрибут *внешконт*. ■

Съемный объект. Указатель, приводящий к объекту контекста съемных носителей, можно задать с помощью имени из метки объекта (см. атрибуты *имяфайла, имяконт*). При этом ис-

пользуется ⟨внешнее имя⟩, начинающееся с ⟨идентификатора⟩ константы или переменной, значением которой является указатель, установленный на корневой справочник контекста съемных носителей (см. п. 5.1). В простейшем случае — это константа *ксн*. Например, к контейнеру с именем "*моймд*" можно обращаться с [помощью ⟨внешнего имени⟩ *ксн*//*моймд*.

■ О б ъ е к т п р о и з в о л ь н о г о к о н т е к с т а. ⟨Внешнее имя⟩ может начинаться с ⟨идентификатора⟩ константы или переменной (отличной от формального параметра), значением которой является некоторый справочник X. В этом случае ⟨внешнее имя⟩ обозначает указатель, установленный на элемент справочника, находящегося во внешнем контексте, корневой справочник которого есть X. Поиск нужного элемента в этом контексте производится, как описано выше.

Э л е м е н т СВС. ⟨Внешнее имя⟩ с ⟨планом поиска⟩, имеющим вид //[⟨выражение⟩], обозначает указатель, установленный на элемент СВС некоторого файла. ⟨Идентификатор⟩ в этом случае задает файл, а значение ⟨выражения⟩ — номер элемента в СВС этого файла. При опущенном ⟨идентификаторе⟩ рассматривается файл объектного кода данной программы.

⟨С т а н д а р т н ы е с л о г и⟩ интерпретируются следующим образом:

— *.обкт* равносилен применению операции *обкт* к предшествующей ему части ⟨внешнего имени⟩, т. е. данный слог приводит к тому объекту, на который установится подвижный указатель, если к нему применить операцию: *обкт* (*ув*, *s*), где *ув* — некоторый подвижный указатель, *s* — часть ⟨внешнего имени⟩, предшествующая рассматриваемому слогу;

— *.текст* (*.код*, *.прогр*). Часть ⟨внешнего имени⟩, предшествующая такому слогу, должна приводить к файлу объектного кода (файлу текста, файлу, содержащему программу). В этом случае слог приводит к элементу СВС файла, содержащему ссылку на текст программы (к элементу СВС файла, содержащему ссылку на код программы, к программе).

О г р а н и ч е н и я. Один ⟨слог⟩ не должен содержать больше 17 букв и/или цифр. ⟨Многослоговое имя⟩ не должно содержать пробелов. ■]

6. Средства работы с мл-контейнером

6.1. Текущая позиция. Текущая позиция в мл-контейнере — это положение головки записи/считывания по отношению к началу контейнера. Файл (зона), на который установлена головка, называется текущим файлом (текущей зоной). Текущая позиция характеризуется значениями позиционных атрибутов контейнера *позконт*,

файлконт, *имяфайла* и позиционным атрибутом файла *блокфайла*. При модификации позиционного атрибута текущая позиция перемещается. Названные атрибуты контейнера имеют следующий смысл:

позконт — этот атрибут может принимать следующие значения типа набор: "*начк*" — начало контейнера, "*конк*" — конец контейнера, "начф" — начало файла, "конф" — конец файла.

файлконт — порядковый номер текущего файла относительно начала контейнера. При модификации этого атрибута текущая позиция перемещается на файл, номер которого равен новому значению атрибута. Если контейнер уже смонтирован на устройство, то в зависимости от исходного местоположения головки относительно файла текущая позиция может установиться как на начало, так и на конец файла. В противном случае файл устанавливается на начало.

имяфайла — имя из метки текущего файла. Если атрибуту присвоить литерный набор или литерную строку, то текущая позиция перемещается на файл с заданным (этим набором или строкой) именем. Текущая позиция внутри файла определяется, как и в случае модификации атрибута *файлконт*.

Конструкция (модификация атрибутов) допускает одновременное изменение нескольких позиционных атрибутов. В частности, для избежания неопределенности в установке текущей позиции можно одновременно с атрибутами *файлконт*, *имяфайла* задавать атрибут *позконт*. Например, модификация:

запатр (*млк*1, *файлконт* : 10, *позконт* : "*начф*"). . .

устанавливает текущую позицию контейнера *млк*1 на начало файла с номером 10.

6.2. Создание файла. Новый файл всегда создается в текущей позиции контейнера. В связи с этим в общем случае для создания файла необходимо предварительно установить нужную позицию, а затем уже произвести генерацию объекта. В качестве основы генератора подается контейнер. В зависимости от исходной позиции различаются 4 случая:

1. Начало контейнера — создается первый файл контейнера.
2. Конец контейнера — создается файл, непосредственно следующий за последним из уже существующих файлов контейнера.
3. Начало файла — старый файл затирается, и на его месте создается новый.
4. Конец файла — создается файл, непосредственно следующий за текущим.

Вновь созданный файл считается последним в контейнере. Текущая позиция устанавливается на начало нового файла.

6.3. Доступ к файлу. Файл, существующий в контейнере, можно открыть, задав значения позиционных атрибутов в операции открытия файла, например,

файл (*млк*1, *файлконт*: 10, *позконт*: "*начф*")

Кроме того, можно открыть файл, предварительно установив на него указатель внешнего объекта. Это можно сделать несколькими способами:

1. Задать постоянный указатель с помощью внешнего имени *ксн*//*s*1//*s*2, где *s*1 — имя из метки контейнера, а *s*2 — имя из метки файла, или *x*//*s*2, где *x* — контейнер.

2. Установить подвижный указатель с помощью операции поиска элемента справочника *элспр*. В операцию подается контейнер (в качестве справочника, см. п. 5.1) и имя искомого файла. В этом случае текущая позиция устанавливается на нужный файл уже при выполнении операции поиска *). Результирующая позиция определяется, как описано выше для случая модификации атрибута *файлконт*.

3. Установить подвижный указатель с помощью операции позиционирования справочника. Как и в предыдущем случае, аргументом операции является контейнер. В процессе выполнения операции текущая позиция устанавливается на нужный файл. Результирующая позиция определяется, как описано выше для случая модификации атрибута *файлконт*.

4. Модифицируя позиционные атрибуты контейнера, установить текущую позицию на начало или на конец нужного файла. Затем, используя операцию копирования указателя *копирув*, установить в эту же позицию указатель внешнего объекта, например:

копирув (*ув*, *запатр* (*млк*1, *файлконт* : 10)),

где *ув* и *млк*1 — соответственно указатель и контейнер.

Указатель, полученный одним из перечисленных способов, (постоянный или подвижный) установлен на закрытый мл-файл. Чтобы открыть файл, указатель надо подать в операцию открытия, например:

млф := *файл* (*ув*)

Если здесь используется постоянный указатель, то сначала производится поиск нужного файла, а затем его открытие. Текущая позиция первоначально устанавливается так, как описано выше для случая модификации атрибута *файлконт*.

6.4. Закрытие файла. При перемещении текущей позиции с одного файла на другой первый автоматически закрывается (если он был открыт).

*) Для мл-контейнера не могут одновременно существовать два подвижных указателя, установленных в разные позиции.

Перемещение текущей позиции, происходящее при закрытии файла, зависит от атрибута *направление*. Значение истина (ложь) задает прямое (обратное) направление обработки, и текущая позиция перемещается на конец (на начало) закрываемого файла.

7. Обмен. Общие сведения

В табл. 1 указаны возможные способы организации обмена. В правой колонке таблицы на примере операции чтения представлен образец конструирования операции для каждого конкретного способа обмена.

Таблица 1

Способы двоичного обмена

обмен	буферизованный	однобуферный	последовательный		*чит (пзп, след (к))*
			произвольный	синхронный	*чит (пзп, нбл)*
				параллельный	*читпар (пзп, нбл)*
		многобуферный	последовательный		*чит (тбф, бдф, след (к))*
			произвольный	синхронный	*чит (тбф, нбл)*
				параллельный	*читпар (тбф, нбл)*
	непосредственный	ацд, пм, ацпу мл, пк, пл	последовательный	синхронный	*чит (ф, след(к), м)*
				параллельный	*читпар (бвф, след (к), м)*
		мб, мд	произвольный	синхронный	*чит (ф, а, м)*
				параллельный	*читпар (бвф, а, м)*

Сокращения: — послед. — последовательный доступ; — произв. — произвольный доступ; — синхр. — синхронный режим обмена; — парал. — параллельный режим обмена.

7.1. Операции обмена. Конструкция ⟨двоичный обмен⟩ предназначена для осуществления операций ввода информации с файла, вывода информации на файл и позиционирования файла. В отличие от ⟨форматного обмена⟩ эти операции не производят редактирование данных.

Ниже описываются наиболее общие свойства операций и даются ссылки на параграфы, в которых представлена подробная семантика для каждого конкретного способа обмена.

⟨двоичный обмен⟩ ::=
⟨идентификатор операции⟩ (⟨выражение⟩,
⟨спецификация адреса⟩ {, ⟨выражение⟩})
⟨идентификатор операции⟩ ::=
чит | *читпар* | *зап* | *заппар* | *нов* | *уст* | *устпар*
⟨спецификация адреса⟩ ::= ⟨выражение⟩

| ⟨направление⟩ { (⟨выражение⟩)}
| ⟨выражение⟩, ⟨направление⟩ { (⟨выражение⟩)}
| *нач* | *кон* | *канал* (⟨выражение⟩)
⟨направление⟩::= *след* | *пред*

Способы обмена. Значением первого ⟨выражения⟩ может быть:

— позиционная переменная. В этом случае выполняется одна из операций однобуферного способа буферизованного обмена (см. § 10);

— таблица буферов. В этом случае выполняется одна из операций многобуферного способа буферизованного обмена (см. § 11);

— заголовок открытого файла. Выполняется одна из операций синхронного непосредственного обмена (см. § 8);

— блок ввода/вывода. В этом случае выполняется одна из операций параллельного непосредственного обмена (см. § 8).

Обозначение операции. ⟨Идентификатор⟩ операции задает операцию, выполняемую над объектом:

— *чит*, *читпар* — операции чтения (синхронного, параллельного);

— *зап*, *заппар* — операции записи (синхронной, параллельной);

— *нов* — дополнительная операция записи без предварительного считывания блока в буфер;

— *уст*, *устпар* — операции позиционирования (синхронного, параллельного).

Тип доступа. Существуют операции для последовательного доступа и операции для произвольного доступа. Тип доступа (последовательный, произвольный) определяет ⟨спецификация адреса⟩. Там же указывается участок файла, подлежащий обработке:

— форма ⟨выражение⟩ используется в операциях произвольного доступа. Значение ⟨выражения⟩ задает адрес начала обмена. Такой тип доступа возможен для мб- и мд-файлов (см. пп. 8.1, 10.3, 11.1) и для мл-файлов при буферизованном обмене (см. п. 10.3);

— форма *след* (⟨выражение⟩) используется в операциях последовательного доступа. Значение ⟨выражения⟩ задает смещение текущей позиции. Такой тип доступа возможен для физически последовательных файлов (см. пп. 8.3—9.6) и при буферизованном обмене для файлов всех типов (см. § 10). Если указано ⟨направление⟩ *пред*, то обработка производится в обратном направлении (от конца файла к началу). Если скобки и ⟨выражение⟩ опущены, то подразумевается смещение, равное 1;

— форма из двух компонентов ⟨выражение⟩, *след* (⟨выражение⟩) используется в операциях последовательного доступа при многобуферном способе буферизованного обмена (см. п. 11.2). В этом

способе считается, что существует несколько текущих позиций, и с каждой из них связан буфер. Значением первого ⟨выражения⟩ является один из этих буферов, а значение второго ⟨выражения⟩ задает смещение текущей позиции, соответствующей этому буферу. Если указано направление *пред*, то обработка производится в обратном направлении. Если скобки и второе ⟨выражение⟩ опущены, то подразумевается смещение, равное 1;

— формы *нач*, *кон* задают в операциях позиционирования смещение текущей позиции на начало файла и на конец файла (см. пп. 8.3, 8.4, 10.1, 11.2);

— форма *канал* (⟨выражение⟩) представляет собой специфическое указание позиции, предназначенное только для ацпу-файлов (см. пп. 8.5, 10.1).

Массив обмена. При непосредственном обмене (см. § 8) в операциях чтения/записи необходимо задавать массив обмена. В данном случае допускается только вектор, который является значением ⟨выражения⟩, следующего за ⟨спецификацией адреса⟩. В операциях чтения информация считывается из файла в этот вектор, а в операциях записи информация из этого вектора записывается в файл. При буферизованном обмене эта компонента не указывается, так как обмен происходит с буфером.

Результат. Операции, отличные от операций буферизованного чтения (записи), в качестве результата выдают первый параметр. Операции буферизованного чтения/записи выдают буферный массив.

7.2. Назначение буферизованного обмена. При буферизованном способе обмена взаимодействие с памятью файла осуществляется с помощью промежуточных буферов. При этом считается, что файл логически состоит из блоков, упорядоченных по номерам. В зависимости от типа внешнего устройства различаются файлы, состоящие из блоков постоянной длины, и файлы, состоящие из блоков переменной длины.

Однобуферный обмен. Существуют два способа буферизованного обмена: однобуферный и многобуферный. Однобуферный способ в основном предназначен для реализации традиционного последовательного доступа. В этом случае с файлом связана позиционная переменная. Позиционная переменная представляет собой объект связи с файлом и создается с помощью генератора, например:

*поз*1 := *позп* (*мдф*1),

здесь создается позиционная переменная для буферизованного обмена с файлом *мдф*1;

*поз*2 := *позп* (//*файл*2),

здесь создается позиционная переменная для буферизованного обмена с файлом, доступным по внешнему имени//*файл*2.

Считается, что при буферизованном обмене с файлом существует текущая позиция, характеризующая положение головки записи/считывания по отношению к началу файла. Головка установлена на текущий обрабатываемый блок. Позиционная переменная содержит информацию о текущей позиции, а именно: значение ее атрибута *блокфайла* — это номер текущего блока.

С позиционной переменной связан текущий буфер. Этот буфер является отображением текущего блока в оперативной памяти. В результате выполнения операции чтения или записи текущая позиция сдвигается на следующий (или предыдущий — в зависимости от направления обработки) блок. Этот блок становится текущим, и выдается вектор для считывания из буфера содержимого текущего блока или для записи в буфер текущего блока, например:

*буф*1 := *чит* (*поз*1, *след*)
*буф*2 := *зап* (*поз*2, *след*)

Используя полученный вектор, можно записывать или считывать значения элементов обычным образом:

*x*1 := *буф*1 [3] % считывание из буфера
*a*1 <:= *буф*1 **длиной** *к* % считывание из буфера
*буф*2 [*i*] := *x*2 % запись в буфер
*буф*2 <:= *a*2 % запись в буфер

Операция чтения всегда производит считывание соответствующего блока в буфер. В отличие от этого, операция записи производит считывание блока только в том случае, если он находится в пределах заполненной части файла. Если же в заполненной части файла необходимо полностью заменить некоторый блок, т. е. нет необходимости считывать его старое содержимое, то для этого нужно использовать другую операцию записи:

дз := *нов* (*поз*2, *след*)

В этом случае выдается вектор для записи, но блок не считывается в буфер.

Если в буфер производилась запись, то по окончании работы с этим буфером система осуществляет автоматический сброс содержимого буфера в файл.

После того как произошел сдвиг текущей позиции с одного блока на другой, доступ к буферу первого становится невозможен. Таким образом в данном способе в каждый момент можно иметь доступ только к одному, а именно текущему блоку файла.

Многобуферный обмен. Многобуферный способ буферизованного обмена осуществляется через объект «таблица

буферов». Эта таблица создается с помощью генератора, например:

*многобуфмдф*1 := *тбуф* (*мдф*1)
*многобуфмдф*2 := *тбуф* (//*файл*2)

В многобуферном способе не существует специальной позиционной переменной. Позиция указывается с помощью номера блока.

Как и в однобуферном способе, в результате обращения к некоторому блоку выдается буфер этого блока, но при этом не ликвидируется возможность доступа к буферам, которые обрабатывались перед этим. Например, последовательность двух операторов:

*дбл*1 := *чит* (*многобуфмдф*1, *нбл*1)
*дбл*2 := *чит* (*многобуфмдф*1, *нбл*2)

дает возможность одновременно работать с двумя буферами, а именно: с помощью *дбл*1 — с буфером, соответствующим блоку *нбл*1, и с помощью *дбл*2 — с буфером, соответствующим блоку *нбл*2. Каждый буфер остается заблокированным до тех пор, пока программа не осуществит его явную разблокировку, например:

открепбуф (*многобуфмдф*1, *дбл*1)

Функции системы при буферизованном обмене. Система обеспечивает создание, уничтожение и переиспользование буферов, а также предварительное их заполнение блоками файла (параллельно с выполнением программы) при последовательном доступе. ■

7.3. Назначение непосредственного обмена. Процедуры непосредственного обмена дают возможность реализовать пересылку данных между памятью программы и файлом без какого-либо промежуточного хранения. Непосредственный обмен можно осуществлять как синхронно с работой программы, так и параллельно.

■ *Режим обмена.* Семантическое отличие синхронного непосредственного обмена от параллельного заключается в следующем. В первом случае к моменту завершения выполнения операции обмена соответствующая операция над внешним носителем уже выполнена. Во втором случае операция только инициирует начало обмена. В случае строго последовательных внешних устройств (мл, пк, ацпу, пл) при параллельном обмене сохраняется порядок выполнения реальных операций, но по времени они в общем случае запаздывают по отношению к процессу выполнения программы. При необходимости это запаздывание можно ликвидировать, используя в программе операции синхронизации над семафором завершения обмена.

Блок ввода/вывода. Одной из основных функций системы при реализации некоторой конкретной пересылки данных между опе-

ративной памятью и файлом является создание так называемого блока ввода/вывода (БВВ). БВВ, как и позиционная переменная, является объектом связи с файлом. Назначение БВВ состоит в том, что он связывает файл и тот массив в оперативной памяти, с которым необходимо произвести обмен. Кроме того, в БВВ хранится информация об операции, о местоположении обрабатываемого участка файла, о реакциях на возможные ошибки обмена и пр.

С точки врения параметров операций различие между синхронным и параллельным непосредственным обменом состоит в следующем. В первом случае система неявно формирует БВВ на время данного обмена, и первым параметром процедуры обмена является заголовок открытого файла, например:

*диск*1 := *файл* (//*файл*2)

Здесь открывается файл, доступный по внешнему имени//*файл* 2, и значением переменной *диск*1 становится заголовок открытого файла;

чит (*диск*1, *адр*1, *мас*1),

здесь информация из файла, начиная с адреса *адр*1, читается в массив *мас*1.

Во втором случае сама программа должна явно создать БВВ с помощью генератора, например:

*бввдиск*2 := *бвв* (//*файл*2)

и использовать его в качестве первого параметра процедуры обмена:

читпар (*бввдиск*2, *адр*2, *мас*2)

Одним из атрибутов БВВ является семафор завершения обмена. Запросив этот атрибут, можно реализовать синхронизацию с процессом обмена:

ждать (имя *читатр* (*бввдиск*2, *смфобмена*) @) ■

7.4. Обозначения, используемые при описании операций. В описании семантики операций используются следующие обозначения для параметров:

ф (*пзп*, *тбф*, *бвф*) — заголовок открытого файла (позиционная переменная, таблица буферов, блок ввода/вывода);

буф — буферный массив, над которым выполняется операция;

а — номер сектора мб- или мд-файла в операциях непосредственного обмена;

нбл — номер блока в операциях буферизованного обмена;

м — массив в операциях непосредственного обмена;

к — величина смещения текущей позиции.

Кроме того, используются следующие обозначения:

блкф — номер первого незаполненного блока файла;

пэп'блокфайла — атрибут *блокфайла* из позиционной переменной *пзп*. Аналогичным образом обозначаются другие атрибуты позиционной переменной, файла, ББВ и буферного массива.

8. Непосредственный обмен

Наличие специфических особенностей у внешних устройств различного типа приводит к тому, что трактовка операций непосредственного обмена зависит от типа устройства, на котором расположен файл. В связи с этим описание непосредственного обмена ориентировано на уровень внешних устройств и разбито на пункты, соответствующие типам устройств.

8.1. **Мб- и мд-файлы.** Ввиду того, что принципы организации и средства обмена с мб-файлами и мд-файлами схожи между собой, в последующем описании, если нет необходимости их различать, будет употребляться термин «мбмд-файл» и «мбмд-носитель».

Физическая адресация к мбмд-носителю осуществляется с точностью до сектора. Физический объем сектора мб-носителя — 32 72-разрядных слова, а сектора мд-носителя определяется значением атрибута *длинсектора*.

Для реализации распределения физической памяти на мбмд-носителе используется метод листового распределения памяти, т. е. математическим листам файла ставятся в соответствие физические листы. Физический лист представляет собой совокупность нескольких смежных секторов. Все математические листы некоторого конкретного файла упорядочены по номерам и имеют одинаковую длину, равную длине физического листа этого файла. Длина физического листа определяется значением атрибута *длинлиста* для данного файла (для разных файлов на данном носителе это значение может быть различно). Значение этого атрибута задает количество секторов в листе. Максимально возможное количество листов файла определяется значением атрибута *максдлинфайла*.

■ В общем случае физические листы некоторого файла располагаются на носителе несмежно. Отведение физического листа, соответствующего некоторому математическому листу, осуществляется динамически, в момент первой записи в этот лист. Однако с помощью атрибутов *порядоклистов*, *листвцилиндр*, *класслиста* и *файлвпакет* можно управлять расположением физических листов на носителе, а с помощью атрибута *распределен* — отведением и освобождением физических листов.

Операция *перестлист* (*ф1*, *ф2*, *k1*, *m1*,. . ., *ki*, *mi*,. . .) переставляет местами лист с номером *ki* из файла *ф1* и лист с номером *mi* из файла *ф2*. При этом физической переписи листов не происходит, а осуществляется перестановка базовых адресов листов в таблицах

листов указанных файлов. На параметры накладываются следующие ограничения: значения атрибута *длинлиста* у обоих файлов должны совпадать: оба файла должны либо быть мб-файлами, либо принадлежать одному и тому же мд-контейнеру. В частности, *ф*1 и *ф*2 могут являться одним и тем же файлом. ■

Операции пересылки данных. В случае непосредственного обмена мбмд-файлы рассматриваются как файлы с произвольным доступом. При этом считается, что элементы файла — это элементы его секторов. Адрес по файлу задается с точностью до сектора, т. е. адрес — это математический номер сектора относительно начала файла. Количество считываемой или записываемой информации не обязательно должно быть кратно сектору. Однако обмен порциями, кратными сектору, позволяет более рационально использовать память на носителе.

Параметры операций. Параметрами операций синхронной пересылки данных являются: заголовок открытого файла (*ф*), адрес начала обмена (*а*) и массив обмена (*м*). Массив обмена может быть представлен:

1. Вектором элементов формата ф64.

■ 2. Массивом векторов элементов формата ф64 (только в привилегированном режиме). В этом случае обмен производится последовательно для каждого из векторов. Параметр *м* в этом случае является целым числом, задающим адрес массива.

В непривилегированном режиме накладываются следующие ограничения:

1. Вектор не может располагаться в стеке.

2. Один и тот же вектор не может одновременно участвовать в двух операциях обмена.

Нарушение ограничений приводит к возникновению «ошибки в параметрах обмена».

Ниже в квадратных скобках указывается форма вызова операций запуска параллельного обмена. Значением первого параметра в данном случае является блок ввода/вывода (*бвф*). К этому блоку предварительно должен быть прикреплен файл, над которым и будет осуществляться операция. Смысл остальных двух параметров совпадает с аналогичными параметрами синхронного обмена. Семафор завершения обмена можно получить, запросив значение атрибута *смфобмена*. ■

1. *чит* (*ф*, *а*, *м*) [*читпар* (*бвф*, *а*, *м*)]

В вектор *м* считывается содержимое последовательности элементов файла, начинающейся от начала сектора *а*. Количество считываемых элементов равно длине вектора *м*. Тип результирующих значений элементов вектора *м* определяется значением атрибута *типданных*.

В качестве результата выдается значение *ф* [*бзф*].

■ Если адрес начала чтения приходится на математический лист файла, для которого еще не отведен физический лист, то возникает ошибка «нет листа».

Если адрес начала чтения выходит за пределы максимально возможного объема памяти файла, то возникает ошибка «физический конец файла». ■

2. *зап* (*ф*, *а*, *м*) [*заппар* (*бвф*, *а*, *м*)]

Значения элементов вектора *м* переписываются в файл, в последовательность элементов, начинающуюся с сектора *а*. Количество заполненных элементов равно длине вектора *м*. Если оказывается, что последний сектор прописан не до конца, то его остаток заполняется нулями. В зависимости от значения атрибута *типданных* информация о типе записываемых значений сохраняется или теряется.

В качестве результата возвращается значение *ф* [*бвф*].

■ Если адрес начала записи выходит за пределы максимально возможного объема файла, то возникает ошибка «физический конец файла». ■

8.2. Общие свойства последовательного доступа. Ниже описываются файлы с последовательным доступом. Такой файл состоит из последовательности упорядоченных по номерам физических записей (зоны мл-файла, перфокарты пк-файла, строчки ацпу, группы символов пл-файла). Перед началом операции обмена головка записи/считывания находится между записями. Запись, предшествующая головке, т. е. только что обработанная (направление обработки — от начала файла к его концу), называется текущей. Номер текущей физической записи задается значением атрибута *блокфайла*. Каждая очередная операция обмена производит действия над записью, следующей за текущей. В результате выполнения операции текущая позиция перемещается. Соответственно корректируется значение атрибута *блокфайла*: при записи/чтении оно становится равным номеру записанной/считанной записи, а при позиционировании — номеру последней пропущенной записи. В число операций над файлами с последовательным доступом входят только такие операции, которые приводят к относительному смещению текущей позиции, а не к произвольной установке.

Параметрами операции синхронной пересылки данных являются заголовок открытого файла (*ф*) и вектор байтовых элементов *) (*м*).

■ Попытка чтения или позиционирования за пределами заполненной части файла приводит к ошибке «логический конец файла».

*) Следуя устоявшейся терминологии, в описании операций обмена будет использоваться термин «байтовый формат». Этот формат совпадает с литерным форматом и равен 8 разрядам.

Если какая-либо операция, связанная с выводом информации, приводит к тому, что результирующий размер файла превышает физический объем носителя, то возникает ошибка «физический конец файла».

В случае параллельного обмена отличие структуры операции состоит в том, что первым параметром должен быть блок ввода/вывода (как и при произвольном доступе). Необходимо учитывать, что при параллельном обмене текущая позиция — это та, в которую головка записи/считывания установлена непосредственно перед началом реального выполнения внешним устройством операции пересылки данных или операции позиционирования, а не в момент инициирования операции программой. ■

8.3. Мл-файлы. Записью мл-файла является зона. Длины различных зон одного файла могут отличаться. Зона состоит из последовательности байтовых элементов, упорядоченных по номерам. Количество элементов в зоне может изменяться от 18 до 999 999. Зоны отделяются одна от другой межзонным промежутком. Начальной зоне файла непосредственно предшествует начальная метка мл-файла, а за конечной зоной следует конечная метка. Эти метки представляют собой отображение заголовка мл-файла.

Текущая позиция характеризуется значением позиционного атрибута *блокфайла* (номер текущей зоны относительно начала файла).

Для мл-файлов возможно позиционирование и чтение не только в прямом, но и в обратном направлении. В этом случае в соответствующей синтаксической конструкции указывается спецификация *пред*. Очередная операция осуществляется над записью, предшествующей головке записи/считывания, т. е. с учетом введенной выше терминологии,— над текущей записью.

Операции позиционирования.

1) *уст* (*ф*, *нач*), [*устпар* (*бвф*, *нач*)]

Текущая позиция устанавливается в положение, следующее за начальной меткой файла.

2) *уст* (*ф*, *кон*) [*устпар* (*бвф*, *кон*)]

Текущая позиция устанавливается в положение, предшествующее конечной метке файла.

3) *уст* (*ф*, *след*(*к*)) при $к \geqslant 1$ [*устпар* (*бвф*, *след*(*к*))]
уст (*ф*, *пред*(*к*)) [*устпар* (*бвф*, *пред*(*к*))]

Пропускается *к* зон файла.

4) *уст* (*ф*, *след*) [*устпар* (*бвф*, *след*)]

уст (*ф*, *пред*) [*устпар* (*бвф*, *пред*)]
уст (*ф*, *след*) эквивалентно *уст* (*ф*, *след*(1))
уст (*ф*, *пред*) эквивалентно *уст* (*ф*, *пред*(1))

Операции пересылки данных.

1) *чит (ф, след, м)* [*читпар (бвф, след, м)*]
чит (ф, пред, м) [*читпар (бвф, след, м)*]

Содержимое элементов зоны считывается в вектор *м*. Считывание в прямом направлении происходит в порядке возрастания номеров элементов зоны и вектора, начиная от 0. Считывание в обратном направлении происходит в порядке убывания номеров элементов, начиная от последних элементов зоны и вектора.

Если в векторе *м* меньше элементов, чем в зоне, то значением атрибута *переполнен* становится 1. Если в векторе *м* больше элементов, чем в зоне, то значением этого атрибута становится —1, и вектор *м* заполняется не до конца (при обратном направлении — не до начала). Если число элементов в зоне и в векторе совпадают, то значение этого атрибута становится нулевым.

2) *зап (ф, сле', м)* [*заппар (бвф, след, м)*]

Содержимое элементов вектора *м* записывается во вновь созданную зону файла. Длина этой зоны равна длине вектора *м*. Старая зона, которой соответствовал такой же номер (если такая была), и зоны с бо́льшими номерами (если такие были), а также все файлы, следующие за данным в контейнере (если такие были), ликвидируются. Новая зона считается последней зоной файла.

Если над файлом после операции записи выполняется какая-либо операция, отличная от записи, то этой последней операции предшествует запись на ленту конечной метки файла и признака конца контейнера.

8.4. Зпк- и чпк-файлы. В последующем описании, если нет необходимости различать зпк- и чпк-файлы, употребляется термин пк-файлы.

Записью пк-файла является перфокарта. Перфокарта состоит из последовательности 12-разрядных элементов (колонок). Каждая перфокарта содержит 80 колонок. Начальной перфокарте пк-файла предшествует перфокарта, содержащая начальную метку пк-файла. Эта метка представляет собой отображение заголовка пк-файла. За конечной перфокартой памяти пк-файла следует перфокарта, содержащая конечную метку пк-файла. Структура меток пк-файла приведена в Приложении 4.

Текущая позиция характеризуется значением атрибута *блок-файла* (номер текущей перфокарты).

Операции позиционирования.

уст (ф, кон) [*устпар (бвф, кон)*]

Операция предназначена для чпк-файлов. Начиная с перфокарты, следующей за текущей, пропускаются все перфокарты чпк-фай-

ла. Кроме того, пропускается перфокарта, содержащая конечную метку пк-файла.

Операции пересылки данных.

1) *чит* (*ф*, *след*, *м*) [*читпар* (*двф*, *след*, *м*)]

Операция предназначена для чпк-файлов. Значения элементов перфокарты перекодируются и считываются в вектор *м*.

Если в векторе *м* меньше, чем 80 элементов (байтовых или 16-разрядных), то информация, находящаяся в старших колонках перфокарты, пропадает, и значение атрибута *переполнен* становится равным 1. Если в векторе *м* больше, чем 80 элементов, то вектор заполняется не до конца.

Возможны два варианта перекодировки:

А. Если значение атрибута *двоичный* есть **ложь**, то считается, что вектор *м* состоит из байтовых элементов, а информация на перфокарте представлена в коде КПК-12. Каждому 12-разрядному элементу перфокарты ставится в соответствие его 8-разрядное представление в коде ДКОИ-8. Это 8-разрядное представление и рассматривается как результат перекодировки. Если значение какого-либо 12-разрядного элемента перфокарты таково, что для него не существует соответствующего 8-разрядного эквивалента, то возникает «ошибка символа».

Б. Если значение атрибута *двоичный* есть **истина**, то считается, что вектор *м* состоит из 16-разрядных элементов, а информация на перфокарте представлена в двоичном виде. Перекодировка значения элемента перфокарты состоит в том, что к левым и к правым 6 разрядам добавляется по два старших нулевых разряда. В результате образуется 16-разрядное значение, которое и рассматривается как результат перекодировки.

2) *зап* (*ф*, *след*, *м*) [*заппар* (*двф*, *след*, *м*)]

Операция предназначена для зпк-файлов. Значения элементов вектора *м* (8- или 16-разрядные, в зависимости от атрибута *двоичный*) перекодируются в 12-разрядные значения. Последовательность полученных значений выводится в виде новой перфокарты файла.

Если в векторе *м* больше чем 80 элементов, то используются только первые 80. Если в векторе *м* меньше чем 80 элементов, то соответствующее количество колонок перфокарты со старшими номерами не будет содержать пробивок.

Возможны два варианта перекодировки:

А. Если значение атрибута *двоичный* есть **ложь**, то считается, что вектор *м* состоит из байтовых элементов, и значение каждого элемента перекодируется из кода ДКОИ-8 в 12-разрядное значение в коде КПК-12.

Б. Если значение атрибута *двоичный* есть истина, то считается, что вектор *м* состоит из 16-разрядных элементов. Перекодировка значения элемента вектора состоит в том, что в левых и правых разрядах значения убираются по 2 старших разряда. Образовавшееся 12-разрядное значение и рассматривается как результат перекодировки.

8.5. **Ацпу-файлы.** Записью ацпу-файла является строчка на бумажном носителе. Строчка состоит из последовательности литер, максимальная длина строки определяется значением атрибута *максдлинстроч*. Строчка печатается, начиная с самой левой печатной позиции носителя. Начальной строчке памяти файла предшествует начальная метка ацпу-файла, а за конечной строчкой следует конечная метка ацпу-файла. Эти метки представляют собой отображение заголовка файла.

Текущая позиция характеризуется значением атрибута *блокфайла*. Если программа не осуществляет управление вертикальной табуляцией носителя (с помощью операции «прогон к каналу»), то значение этого атрибута в точности соответствует расположению печатающейся строчки относительно начала файла. В противном случае вертикальное расположение печатающейся строчки определяется программой.

Отличное от нуля значение атрибута *максдлинстран* задает размер логической страницы ацпу-файла. В этом случае система осуществляет следующие действия по разбиению текста на страницы. Если значение параметра *к* больше 0 (см. ниже операции позиционирования и пересылки данных), то при условии, что *ф'строчка* $<$ $<$ *ф'максдлинстран*, в конце операции значение атрибута *строчка* корректируется: *ф'строчка* := *ф'строчка* + *к*; если же последнее условие не выполняется, то в конце операции атрибут *строчка* становится равным *к*, и значение атрибута *страница* увеличивается на 1. Операция «прогон к каналу» (см. операции позиционирования и передачи данных) не изменяет значение атрибута *строчка*, кроме случая прогона к каналу с номером 1. В этом случае значение атрибута *строчка* становится равно 1.

Операции позиционирования.

1) *уст* (*ф*, *след*(*к*)) при $к \geqslant 1$ [*устпар* (*бвф*, *след*(*к*))]

Выполняется вертикальная табуляция бумажного носителя на *к* интервалов, т. е. выводится *к*—1 пустых строчек. Носитель устанавливается на текущую строчку.

2) *уст* (*ф*, *след*) [*устпар* (*бвф*, *след*)]

Операция эквивалентна *уст*(*ф*, *след*(1)).

3) *уст* (*ф*, *канал* (*к*)) [*устпар* (*бвф*, *канал*(*к*))]

Операция осуществляет вертикальную табуляцию бумажного носителя. В результате носитель будет продернут до строчки, номер которой соответствует каналу с номером *к*. Например,

уст (*ф*, *канал*(1))

обычно реализует переход на начало следующей страницы бумажного носителя.

Операции пересылки данных. Семантика операций записи в ацпу-файл отличается от семантики аналогичных операций для других файлов тем, что установка новой позиции происходит после печати новой строчки, а не предшествует ей. Кроме того, используя непосредственный обмен, надо учитывать, что непосредственно после создания ацпу-файла текущая позиция установлена на первую строчку файла, а не предшествует ей.

1) *зап* (*ф*, *след*(*к*), *м*) при *к*=0 [*заппар* (*бвф*, *след*(*к*), *м*)]

Выполнение операции состоит в следующем. Выводится текущая строчка с номером, равным текущему значению атрибута *блок-файла*. Значения элементов вектора *м* становятся значениями соответствующих элементов новой строчки. Если длина вектора *м* не превышает *ф'максдлинстроч* элементов, то длина строчки равна длине вектора *м*. Если вектор *м* содержит более *ф'максдлинстроч* элементов, то печатаются только первые *ф'максдлинстроч* литер.

Примечание. В результате выполнения этой операции текущая позиция не корректируется. Переход на новую строку достигается путем позиционирования, например:

уст (*зап* (*ф*, *след*(0), *м*), *след*).

Если позиционирование не сделано, то последующая печать будет происходить на той же строчке, опять начиная от самой левой печатной позиции.

2) *зап* (*ф*, *след*(*к*), *м*) при *к*≥1 [*заппар* (*бвф*, *след*(*к*), *м*)]

Операция эквивалентна *уст* (*зап* (*ф*, *след*(0), *м*), *след*(*к*)).

3) *зап* (*ф*, *след*, *м*) [*заппар* (*бвф*, *след*, *м*)]

Операция эквивалентна *зап* (*ф*, *след*(1), *м*).

4) *зап* (*ф*, *канал*(*к*), *м*) [*заппар* (*бвф*, *канал*(*к*), *м*)]

Операция эквивалентна *уст* (*зап* (*ф*, *след*(0), *м*), *канал*(*к*)).

8.6. Чпл- и зпл-файлы. В последующем описании, если нет необходимости различать чпл- и зпл-файлы, употребляется термин «пл-файл».

Пл-файл представляет собой последовательность 8-разрядных элементов (или символов) на перфоленте.

Операция пересылки данных осуществляет чтение или запись группы элементов, начинающейся с текущего элемента. Если значение атрибута *кодблока* есть **истина**, то группы разделяются особым символом, не включаемым в группу. Этот символ называется кодом конца блока и набирается на панели устройства.

Текущая позиция характеризуется значением атрибута *блок-файла*, который задает порядковый номер последней записанной или считанной группы символов.

Операции пересылки данных.

1) *чит* (*ф*, *след*, *м*) [*читпар* (*бвф*, *след*, *м*)].

Операция предназначена для чпл-файлов. Группа символов считается в вектор *м*. Длина этой группы (*к*) определяется следующим образом:

— если значение атрибута *кодблока* есть **ложь**, то *к* равно длине вектора *м*;

— если значение атрибута *кодблока* есть **истина**, то *к* равно либо количеству элементов до ближайшего кода конца блока (не включая его), либо длине вектора *м* (если количество элементов до ближайшего кода конца блока больше либо равно длине вектора *м*). Головка считывания устанавливается перед первым необработанным символом (код конца блока автоматически минуется).

■ Если оказалось, что количество элементов до ближайшего кода конца блока (при значении атрибута *кодблока*, равном **истина**) больше либо равно длине вектора *м*, то значением атрибута *переполнен* становится 1, а значением атрибута *длинблока* — длина вектора *м*. В противном случае значением атрибута *переполнен* становится — 1, вектор *м* заполняется не до конца, и значением атрибута *длинблока* становится длина заполненной части этого вектора. ■

2) *зап* (*ф*, *след*, *м*) [*заппар* (*бвф*, *след*, *м*)]

Операция предназначена для зпл-файлов. Содержимое элементов вектора *м* выводится на перфоленту в виде очередной группы символов.

Если значение атрибута *кодблока* есть **истина**, то после группы вновь созданных элементов помещается символ кода конца блока.

8.7. Ацд- и пм-файлы. Записью ацд- и пм-файла является сообщение. Длины различных сообщений одного файла могут отличаться. Сообщение представляет собой последовательность литер. Различаются входные и выходные сообщения. Те сообщения, которые вводятся с клавиатуры устройства на носитель файла (экран ацд-файла, бумажный носитель пм-файла) и затем читаются программой, называются входными. Те сообщения, которые записываются на носитель файла программой, называются выходными.

Текущая позиция файла характеризуется значением атрибута блокфайла (номер текущего сообщения).

О п е р а ц и и п е р е с ы л к и д а н н ы х. При появлении входного сообщения происходит одно из двух:

— программа ожидает появления входного сообщения. В этом случае сообщение сразу же поступает в обработку;

— сообщение появляется до того, как программа его затребует с помощью операции чтения. В этом случае, если значением атрибута *активность* является процедура, то эта процедура запускается как параллельный процесс. Таким образом, с помощью указанного атрибута можно определить программную реакцию на независимую активность со стороны терминала.

1) *чит* (*ф*, *след*, *м*)

При выполнении операции чтения выводится приглашение, если задан режим «с приглашением» (см. атрибут *приглашение*), и программа ожидает вводное сообщение. После появления входного сообщения оно считывается в вектор *м*.

Атрибуты *переполнен* и *длинблока* устанавливаются так же, как и для других последовательных файлов (см., например, мл-файлы).

2) *зап* (*ф*, *след*, *м*)

Содержимое вектора *м* выводится в качестве текущего выходного сообщения. Длина сообщения равна длине вектора *м*.

9. Буферизованный обмен

9.1. Блоки постоянной и переменной длины. Как указывалось выше, в операциях буферизованного обмена считается, что файл логически состоит из последовательности блоков. Элементам этой последовательности поставлены в соответствие номера. Различаются файлы с блоками постоянной длины и файлы с блоками переменной длины. В первом случае файл состоит из блоков, количество элементов в которых равно значению атрибута *максдлинблока*. Во втором случае файл состоит из блоков, количество элементов в которых переменно, но не превышает значения атрибута *максдлинблока*. Длина блока мб- и мд-файла измеряется в словах; на других устройствах длина блока задается в байтах.

Ниже для файлов всех типов приводится соответствие между блоками файла при буферизованном обмене и записями файла при непосредственном обмене.

М б- и м д-ф а й л ы являются файлами с блоками постоянной длины. Последовательность блоков файла располагается на последовательности математических листов файла в порядке возрастания

номеров. Начало блока совпадает с началом сектора. Элементами блока являются элементы файла, попавшие в этот блок.

М л - ф а й л ы являются файлами с блоками переменной длины. Значение атрибута *максдлинблока* не превышает 999 999 байтов. Физически блок с номером *к* представляет собой зону с номером *к*. Элементами блока являются элементы зоны. Минимальная возможная длина блока составляет 18 байтов.

З п к - и ч п к - ф а й л ы являются файлами с блоками постоянной длины. Физически блоки зпк- и чпк-файлов располагаются на 80-колоночных перфокартах. Элементам блока соответствуют колонки перфокарты. Информация располагается на перфокарте, начиная с колонки с номером 1.

Если значение атрибута *двоичный* есть **ложь** (т. е. информация на перфокартах представлена в коде КПК-12), то 8-разрядные значения элементов блока представляют собой перекодированную форму соответствующих 12-разрядных значений элементов (колонок) перфокарты.

Если значение атрибута *двоичный* есть **истина** (т. е. информация на перфокартах представлена в двоичном виде), то блок должен содержать четное число байтов. Если считать, что блок состоит из 16-разрядных элементов, то 16-разрядные значения элементов блока представляют собой перекодированную форму (см. п. 8.4) соответствующих 12-разрядных значений элементов (колонок) перфокарты.

А ц п у - ф а й л ы являются файлами с блоками постоянной длины. Физически блок с номером *к* представляет собой строчку файла с номером *к*. Элементами блока являются элементы строчки. Длина блока не должна превышать значения атрибута *максдлинстроч*, задающего реальную длину строчки на используемом устройстве.

Ч п л - ф а й л ы со значением атрибута *кодблока*, равным значению **ложь**, являются файлами с блоками постоянной длины. Блок такого чпл-файла определяется следующим образом. Элементы чпл-файла последовательно объединяются в группы по *т* элементов в группе, где *т* определяется значением атрибута *максдлинблока*; *к*-я группа представляет собой блок с номером *к*. Элементами *к*-го блока являются элементы *к*-й группы.

Чпл-файлы со значением атрибута *кодблока*, равным значению **истина**, являются файлами с блоками переменной длины. Текущий блок такого файла представляет собой группу символов, участвующую в очередной операции считывания в буферный массив. Длина этой группы определяется, как и в операции чтения чпл-файла при непосредственном обмене (см. п. 8.6).

З п л - ф а й л ы (независимо от значения атрибута *кодблока*) являются файлами с блоками переменной длины. Блоками такого файла являются группы символов, выведенные с помощью операций

записи. После каждого очередного блока зпл-файла со значением атрибута *кодблока*, равным значению **истина**, помещается код конца блока.

А ц д - и п м - ф а й л ы являются файлами с блоками переменной длины. Физически блок с номером *к* представляет собой сообщение с номером *к*. Элементами блока являются литеры сообщения.

9.2. Буфер файла. Два способа буферизованного обмена. В процессе буферизованного обмена используются объекты, которые называются буферами данного файла. Основной компонентой буфера является буферный массив. Этот массив представляет собой вектор элементов, формат которых определяется значением атрибута *формэлем*. Значение атрибута *блокфайла* из буфера представляет собой целое число, равное номеру блока, которому соответствует данный буфер.

Кроме атрибута *блокфайла*, основными атрибутами буфера являются:

1) *длинблока*. Определяет реальную длину считанного или записываемого блока в случае файлов с блоками переменной длины. Для файлов с блоками постоянной длины значение этого атрибута равно значению атрибута *максдлинблока* из файла.

2) *модифицирован*. Определяет, выполнялась ли операция записи в буферный массив (*зап*, *нов*).

■ Когда задача производит чтение или запись некоторого блока файла, создается буфер, соответствующий этому блоку (если такого буфера еще нет). Этот буфер прикрепляется к задаче (или число ссылок на существующий буфер увеличивается на 1). Таким образом, если в данный момент в задаче существует несколько путей буферизованного доступа к одному и тому же блоку некоторого мбмд-файла, то все они осуществляются через один и тот же буфер. Две разные задачи осуществляют буферизованный доступ к одному и тому же файлу через разные комплекты буферов. ■

Операция *чит* считывает содержимое блока в буферный массив и выдает этот массив. Операция *зап* подготавливает буфер, предназначенный для последующей записи в файл, и выдает буферный массив. Если предполагается сделать запись в заполненную часть файла, то при подготовке буфера в него считывается текущее содержимое блока. Программа может заполнять или модифицировать полученный буферный массив. Фактический сброс его содержимого в файл происходит неявно после того, как программа закончила обработку данного блока (точнее — обработку соответствующего буфера).

Операция *чит* выдает часть буферного массива от элемента с номером 0 до элемента с номером *буф'длинблока* — 1. Допускается только считывание значений элементов полученного вектора. По-

пытка записи приводит к возникновению ситуации *нарушениезащиты*.

Операция *зап* выдает весь буферный массив (размер предварительно считанного блока можно узнать, запросив значение атрибута *длинблока*). Допускается считывание и изменение значений элементов полученного вектора.

■ Два способа обмена. Различаются два способа буферизованного обмена: однобуферный и многобуферный. В первом случае с файлом связана позиционная переменная, которая указывает на текущий обрабатываемый блок файла. Когда позиционная переменная смещается на другой блок, происходит автоматическое открепление буфера, соответствующего предыдущему обрабатываемому блоку. Во втором случае такой специальной позиционной переменной не существует. Позиция указывается с помощью номера блока. Начало обработки некоторого блока не приводит к автоматическому откреплению буфера, соответствующего предыдущему обрабатываемому блоку. Открепление буфера осуществляется только явным образом. Таким образом, в этом способе имеется возможность иметь одновременно несколько прикрепленных буферов и тем самым работать одновременно с несколькими блоками файла.

Запись блока. Если в результате открепления буфера (*буф*) оказывается, что на данный буфер не осталось временных ссылок, то такой буфер ликвидируется. Перед ликвидацией проверяется значение атрибута *модифицирован*. Если оно есть **истина**, то происходит сброс содержимого буферного массива на файл. Это означает, что в файл записывается блок с номером *буф'блокфайла*. Длина этого блока определяется значением атрибута *буф'длинблока*. Старый блок с таким же номером (если он есть) ликвидируется. Кроме того, в случае мл-файла ликвидируются блоки с бóльшими номерами (если такие есть) и файлы, расположенные следом за текущим в мл-контейнере (если такие есть). Запись буферного массива в файл выполняется так же, как запись массива при непосредственном обмене. Особенности записи, связанные с типом устройства, см. в § 8.

Ошибки. В начале выполнения операции чтения или записи (в обоих способах буферизованного обмена) производятся следующие проверки логических и физических границ. Если при чтении оказывается, что необходимо считать блок, находящийся за пределами заполненной части файла, то возникает ошибка «логический конец файла». Кроме того, если при чтении мбмд-файла оказывается, что необходимо считать блок, находящийся на математическом листе, для которого не распределен соответствующий физический лист, то возникает ошибка «нет листа». Если в начале выполнения операции записи оказывается, что предполагается сделать запись

блока, находящегося за пределами максимально возможного объема файла, то возникает ошибка «физический конец файла».

Размер заполненной части файла определяется следующим образом. В случае мб- и мд-файлов он задается значением атрибута *блоклкф*, который равен номеру первого незаполненного блока. На других устройствах конец заполненной части файла определяется по характерным физическим признакам носителя: конечная метка мл-файла, конечная перфокарта пк-файла, конец пробивок на перфоленте. В дальнейшем изложении для обозначения номера первого незаполненного блока будет использоваться идентификатор *блкф*.

При записи нового блока в файл может оказаться, что его номер больше *блкф*, т. е. между последним блоком заполненной на текущий момент части файла и новым блоком существуют незаполненные блоки. В этом случае предпринимаются следующие действия:

— мб-, мд-файл. Незаполненные блоки, так же как и новый блок, включаются в заполненную часть файла, т. е. значение атрибута *блоклкф* становится равно: номер нового блока +1. Первоначальное содержимое незаполненных блоков не определено;

— ацпу-файл. Незаполненные блоки выводятся на носитель в виде пустых строчек;

— файлы остальных типов. Возникает ошибка «логический конец файла». ■

10. Однобуферный способ

Напомним, что в этом способе обмена считается, что существует текущая позиция, и номер текущего блока хранится в позиционной переменной. Ниже в описании семантики операций встречается присваивание атрибуту *блокфайла*. Такая форма используется в качестве условного обозначения для описания смещения текущей позиции.

Позиционная переменная имеет дополнительный позиционный атрибут *элемблока*. Программа может придавать этому атрибуту целочисленные значения, уточняя тем самым положение текущей позиции с точностью до элемента текущего блока (см. операции 7, 8 п. 10.2).

Для мб-, мд- и мл-файлов разрешено позиционирование и чтение не только в прямом, но и в обратном направлении. В этом случае в соответствующей синтаксической конструкции указывается спецификация *пред*. Соответствующие этому случаю уточнения в описании семантики операций заключены в квадратные скобки.

■ Для ацпу-файла значение атрибута *максдлинстран*, отличное от нуля, задает размер логической страницы. В этом случае се-

мантика изменения атрибута *пзп'строчка* и *пзп'страница* аналогична той, которая описана для случая непосредственного обмена.

Одновременно несколько позиционных переменных могут существовать только для мб-, мд-, ацд-, пм-файла, причем только в случае мб- и мд-файла они могут работать через один и тот же буфер. В силу этого ограничения замечания в п. 9.2, связанные с одновременной работой нескольких позиционных переменных через один и тот же буфер, имеют отношение только к мб- и мд-файлам. ■

10.1. Операции позиционирования.

1) *уст* (*пзп*, *нач*)

Операция предназначена для мб-, мд- и мл-файлов. Текущая позиция перемещается в начало файла.

Результат: *пзп*.

2) *уст* (*пзп*, *кон*)

Операция предназначена для мб-, мд-, мл-, чпл-, чпк-файлов. Текущая позиция перемещается на конец файла.

Результат: *пзп*.

3) *уст* (*пзп*, *след*(*к*)) при $к \geqslant 1$

Операция предназначена для ацпу-файлов, и производит следующие действия:

пзп' блокфайла := *пзп' блокфайла*+*к*

Результат: *пзп*.

4) *уст* (*пзп*, *след*)

Операция предназначена для ацпу-файлов и эквивалентна:

уст (*пзп*, *след*(1))

5) *уст* (*пзп*, *канал*(*к*)) при $к \geqslant 0$

Дополнительная операция для ацпу-файлов, связанная с вертикальной табуляцией бумажного носителя.

10.2. Операции пересылки данных. Последовательный доступ.

1) *чит* (*пзп*, *след*(*к*)) при $к \geqslant 1$
чит (*пзп*, *пред*(*к*))

Операция предназначена для мб-, мд-, мл-, чпк-, чпл-, ацд- и пм-файлов. Для ацд- и пм-файлов значение *к* должно быть равно 1.

Операция перемещает текущую позицию и выдает буферный массив, содержащий блок, который стал текущим.

Перемещение позиции происходит следующим образом. Если текущая позиция была установлена на начало файла, то

пзп' блокфайла := *к* — 1

а иначе

пзп' блокфайла := *пзп'блокфайла*+*к*,

[Если позиционная переменная установлена на конец файла, то

пзп' блокфайла := *блкф*—*к*,

а иначе

пзп' блокфайла := *пзп' блокфайла*—*к*.]

В результате выполнения операции атрибут *элемблока* принимает значение 0.

2) *чит* (*пзп*, *след*) [*чит* (*пзп*, *пред*)]

Операция эквивалентна:

чит (*пзп*, *след*(1)) [*чит* (*пзп*, *пред*(1))]

3) *зап* (*пзп*, *след*(*к*)) при $к \geqslant 1$

Операция предназначена для мб-, мд-, мл-, зпк-, ацпу-, зпл-, ацд- и мл-файлов. Для зпк- и зпл-файлов значение *к* должно быть равно 1.

Операция перемещает текущую позицию и выдает буферный массив, предназначенный для записи блока, ставшего текущим.

Перемещение позиции происходит следующим образом. Если текущая позиция была установлена на начало файла, то

пзп' блокфайла := *к* — 1

Если текущая позиция была установлена на конец файла, то

пзп' блокфайла := *блкф* + *к*

В остальных случаях

пзп' блокфайла := *пзп' блокфайла* + *к*

В результате выполнения операции атрибут *элемблока* принимает значение 0.

■ Примечание. Завершение выполнения операции *зап* не приводит к сбросу содержимого буферного массива на файл. Как указано в п. 9.2, сброс происходит лишь при полном откреплении буфера. Однако имеет место важный частный случай. Если файл прикреплен только к одной позиционной переменной, то выполнение операции *зап* приводит к тому, что перед созданием буфера для следующего блока происходит полное открепление предыдущего буфера и его сброс (параллельно с выполнением программы). ■

4) *зап* (*пзп*, *след*)

Операция эквивалентна *зап* (*пзп*, *след*(1)).

5) *нов* (*пзп*, *след*(*к*))

Эта операция отличается от описанной выше операции записи тем, что первоначальное содержимое буферного массива не определено. Ее использование позволяет избежать считывания блока файла в буферный массив, если в этом нет необходимости.

6) *нов* (*пзп*, *след*)

Операция эквивалентна *нов* (*пзп*, *след*(1)).

7) *чит* (*пзп*, *след*(*к*)) при *к*=0 [*чит* (*пзп*, *пред*(*к*))]

К моменту начала исполнения этой операции блок *пзп'блок-файла* либо уже считан в соответствующий буфер, либо процесс считывания начат, но еще не окончен. В последнем случае операция ожидает завершения считывания. В качестве результата операции выдается часть буферного массива, начинающаяся с элемента с номером *пзп'элемблока* [кончающаяся элементом с номером *пзп'элемблока* — 1].

8) *зап* (*пзп*, *след*(*к*)) при *к*= 0

Данная операция отличается от предыдущей тем, что допускается также запись элементов полученного вектора. Значением атрибута *буф'модифицирован* становится истина.

■ **10.3. Произвольно-последовательный доступ.** Описываемые ниже операции предназначены в основном для использования в режиме произвольно-последовательного доступа к файлу, т. е. когда производится несколько серий последовательной обработки, но каждой серии предшествует установка текущей позиции в новое положение. Операции устанавливают текущую позицию на блок с заданным номером и осуществляют, если надо, его считывание с файла. Все операции определены только для мб-, мд- и мл-файлов.

Операции чтения и записи могут выполняться как в синхронном, так и в параллельном режиме. Особенности выполнения параллельного режима аналогичны тем, которые отмечены для многобуферного обмена (см. § 11). ■

Операции пересылки данных.

чит (*пзп*, *нбл*) [*читпар* (*пзп*, *нбл*)]
зап (*пзп*, *нбл*) [*заппар* (*пзп*, *нбл*)]
нов (*пзп*, *нбл*)

Данные операции отличаются от соответствующих операций последовательного доступа тем, что все действия производятся для блока с номером *нбл*. Текущая позиция перемещается на этот блок, и он становится текущим.

Пример. Синхронизация с помощью операции *чит*.

текуказ := *читпар* (*пзп*, *текблок*) [*текэлем* :];
. . .предварительные действия. . . ;

чит (*пзп*, *след*(0)); % синхронизация
а< := *текуказ* **длиной** *к* % обработка считанного блока.

Пример. Синхронизация с помощью операции *ждать*.

ждать (**имя** *читатр* (*пзп*, *смфобмена*) @)

11. Многобуферный способ

Данный способ буферизованного обмена определен только для мб- и мд-файлов. Значением первого параметра во всех операциях должна быть таблица буферов.

Операции произвольного доступа могут осуществляться как в синхронном, так и в параллельном режиме. Параллельные операции произвольного доступа отличаются от синхронных тем, что только инициируют считывание блока с номером *нбл* в буферный массив (если выполняется операция *заппар* и блок с номером *нбл* находится за пределами заполненной части файла, то, как и в случае других операций записи, предварительное считывание не происходит). На этом исполнение операции заканчивается. Процесс реального считывания блока происходит параллельно с выполнением программы. Эти операции используются в тех случаях, когда необходимо, чтобы процесс считывания происходил на фоне операций, предшествующих началу обработки буферного массива. В качестве значения эти операции выдают буферный массив. Следует учитывать, что перед тем как производить обращение к элементам буферного массива, необходимо выполнить какую-либо синхронизующую операцию, например:

буф := *читпар* (*тбф*, *нбл*);
. . .предварительные действия. . . ;
чит (*тбф*, *нбл*);
а <:= *буф* **длиной** *к*

или явный способ синхронизации:

ждать (**имя** *читатр* (*тбф*, *буф*, *смфобмена*) @)

Операции явного сброса буферов также могут осуществляться как в синхронном, так и в параллельном режиме. Параллельная операция сброса буферов только инициирует процесс записи содержимого буферного массива на файл. Затем управление возвращается в программу, и процесс записи происходит параллельно с выполнением программы.

Форма конструкций параллельного обмена приведена в квадратных скобках рядом с формой для синхронного обмена.

11.1. Произвольный доступ.

1) *чит* (*тбф*, *нбл*) [*читпар* (*тбф*, *нбл*)]

В качестве результата операция выдает буферный массив, содержащий блок *нбл*, причем допускается только считывание элементов полученного вектора. Попытка записи приводит к возникновению ситуации *нарушениезащиты*.

2) *зап* (*тбф*, *нбл*) [*заппар* (*тбф*, *нбл*)]

Операция в качестве результата выдает буферный массив, предназначенный для записи блока *нбл*. Причем допускается запись и считывание значений элементов полученного вектора.

3) *нов* (*тбф*, *нбл*)

Эта операция отличается от *зап* (*тбф*, *нбл*) тем, что первоначальное содержимое буферного массива всегда неопределенно. Ее использование позволяет избежать считывания блока памяти файла в буферный массив, если в этом нет необходимости.

11.2. Последовательный доступ. В пп. 1—8 семантика операций последовательного доступа выражена через ранее определенные операции. Форма эквивалентной операции приведена справа от знака равенства.

1) *чит* (*тбф*, *буф*, *след*(*к*)) при $к \geqslant 1$
   $\equiv$ *чит* (*тбф*, *буф'блокфайла* + *к*)
2) *чит* (*тбф*, *буф*, *след*)
   = *чит* (*тбф*, *буф*, *след*(1))
3) *зап* (*тбф*, *буф*, *след*(*к*)) при $к \geqslant 1$
   = *зап* (*тбф*, *буф' блокфайла* + *к*)
4) *зап* (*тбф*, *буф*, *след*)
   = *зап* (*тбф*, *буф*, *след*(1))
5) *нов* (*тбф*, *буф*, *след*(*к*)) при $к \geqslant 1$
   = *нов* (*тбф*, *буф' блокфайла* + *к*)
6) *нов* (*тбф*, *буф*, *след*)
   = *нов* (*тбф*, *буф*, *след*(1))
7) *чит* (*тбф*, *буф*, *пред*(*к*))
   = *чит* (*тбф*, *буф' блокфайла* — *к*)
8) *чит* (*тбф*, *буф*, *пред*)
   = *чит* (*тбф*, *буф*, *пред*(1))
9) *уст* (*тбф*, *нач*)

В качестве результата операции выдается объект типа вектора, который может быть использован для того, чтобы идентифицировать позицию «начало файла». В семантике операций пп. 1—8 предполагается, что в этом случае

буф' блокфайла = — 1

10) *уст* (*тбф*, *кон*)

В качестве результата операции выдается объект типа вектора, который может быть использован для того, чтобы идентифицировать позицию «конец файла». В семантике операций пп. 1—8 предполагается, что в этом случае

буф^1 *блокфайла* = *ф*^{1} *блоклкф*

11.3. Операции над буферами.

1) *зап* (*тбф*, *буф*, *след*(*к*)) при *к* = 0

Выполнение операции состоит в том, что значением атрибута *буф*^{1}*модифицирован* становится **истина**. В качестве результата операции выдается вектор *буф*, для которого допускается запись и считывание элементов.

2) *открепбуф* (*тбф*, *буф*1, *буф*2,. . .*буфк*)

Операция осуществляет открепление буфера *буф*1 от данной задачи (при этом уничтожается только одна ссылка из задачи на этот буфер). Если теперь оказывается, что данный буфер не прикреплен к задаче, то соответствие между буфером и блоком файла ликвидируется. При этом, если значение *буф*1^1*модифицирован* есть **истина**, то осуществляется сброс содержимого буферного массива. Здесь следует сделать уточнение относительно открепления буфера. Буфер, в отличие от других объектов, при полном откреплении не уничтожается; система может назначить данный буфер для другого блока. Чтобы уничтожить буфер, надо воспользоваться операцией *ликвидбуф*.

Аналогичные действия осуществляются с буферами *буф*2,. . . *буфк*.

3) *открепбуф* (*тбф*)

Операция осуществляет полное открепление всех буферов файла от задачи с уничтожением всех ссылок из задачи на эти буфера. Как и в предыдущем случае, это может привести к сбросу содержимого буферных массивов, но, вообще говоря, не означает ликвидацию буферов.

4) *ликвидбуф* (*тбф*, *буф*1,. . . ,*буфк*)

Операция осуществляет уничтожение буфера *буф*1. При этом, если значение атрибута *буф*1^1*модифицирован* есть **истина**, то осуществляется сброс содержимого буферного массива.

Аналогичные действия осуществляются с буферами *буф*2,. . . *буфк*.

5) *ликвидбуф* (*тбф*)

Операция осуществляет операцию п. 4) над всеми буферами файла, прикрепленными к данной задаче.

6) *сброс* (*тбф*, *буф*1, *буф*2,. . . ,*буфк*)
[*сброспар* (*тбф*, *буф*1, *буф*2,. . . , *буфк*)].

Операция осуществляет сброс содержимого буферных массивов буферов *буф*1, *буф*2,. . . , *буфк* в *файл*. Буфера после сброса не открепляются; соответствие между буферами и связанными с ними блоками продолжает сохраняться. Таким образом, данная операция дает возможность при необходимости приводить в соответствие содержимое буферных массивов и содержимое блоков на внешнем носителе. Операция может осуществляться как в синхронном, так и в параллельном режиме.

7) *сброс* (*тбф*) [*сброспар* (*тбф*)]

Операция осуществляет сброс содержимого всех буферных массивов на файл (подробно см. предыдущую операцию).

12. Структура текстового файла

В этом разделе описывается структура одного из встроенных типов файлов — текстового файла. Текстовый файл представляет собой обычный файл с присущим ему типом носителя, на котором он расположен, и прочими атрибутами, определяющими его характеристики. Особенность текстового файла состоит в том, что вводится ряд соглашений относительно структуры содержимого файла и ряд специфических атрибутов, характеризующих эту структуру.

Структура строки. Считается, что текстовый файл состоит из строк. Если значение атрибута *длинстрок* отлично от нуля, то длина всех строк одинакова и равна значению этого атрибута. В противном случае файл состоит из строк различной длины. Длина строки в обоих случаях не должна превышать значение атрибута *максдлинблока*. Строка представляет собой последовательность байтовых элементов. Строка может состоять из следующих компонент:

— фиксированный номер строки. Наличие этой компоненты зависит от значения атрибута *длинфнс* (см. ниже);

— информационная часть строки;

— байт, кодирующий конец строки, или байт, содержащий длину строки (в зависимости от типа внешнего устройства, см. ниже).

Нумерация строк. Различаются нумерованные текстовые файлы (значение атрибута *длинфнс* отлично от нуля) и ненумерованные текстовые файлы (значение атрибута *длинфнс* равно нулю). Каждая

строка нумерованного файла содержит фиксированный номер этой строки. Фиксированный номер является целым числом, количество цифр в котором определяется значением атрибута *длинфнс* (в последующем описании, если не оговорено особо, под номером строки понимается ее фиксированный номер). Текстовый файл не может содержать двух разных строк с одинаковыми номерами. Строки расположены строго в порядке возрастания номеров, причем номера соседних строк могут отличаться более чем на 1.

Строка ненумерованного файла не содержит своего номера. При распечатке таких файлов применяется сквозная (через 1) нумерация строк. В результате при вставке или удалении строк старая нумерация строк постоянно меняется.

Структура файла. Расположение строк в памяти файла зависит от типа внешнего устройства.

В мб- и мд-файлах каждый блок файла содержит некоторое число полных строк (т. е. строка не переходит с одного блока на следующий). Строки располагаются последовательно, друг за другом. Первый байт строки содержит ее длину. Далее следуют байты, содержащие номер строки (для нумерованных файлов). Если значение атрибута *кодфнс* равно 0, то номер закодирован в двоичном виде, и под него отводится столько байтов, сколько необходимо, чтобы уместить максимальный номер (определяемый значением атрибута *длинфнс*). Если значение атрибута *кодфнс* равно 1, то номер закодирован в литерном виде, и под него отводится *длинфнс* байтов. За номером следуют информационные байты строки. Первый байт строки содержит ее полную длину, т. е. количество байтов, отведенных под номер, плюс длина информационной части, плюс 1. За последней строкой блока следует байт со значением, равным нулю.

В отличие от мб- и мд-файла мл-файл является файлом с блоками переменной длины. Структура строки и расположение строк внутри блока строятся по аналогии с мб- и мд-файлами.

В пк-файлах каждая новая строка начинается с левой крайней колонки карты. Строки текстового файла с плавающей длиной строки должны заканчиваться байтом «конец строки». Номера строк в файле с фиксированной нумерацией кодируются всегда в литерном виде и занимают *длинфнс* колонок. В отличие от мб- и мд-файлов номер строки не обязательно должен располагаться в начале строки. В общем случае позиция начала номера задается значением атрибута *позфнс*.

Пл-файл состоит из последовательности строк, расположенных друг за другом. Строки текстового файла с плавающей длиной строки должны заканчиваться байтом «конец строки». Номера строк в нумерованном файле кодируются всегда в литерном виде, занимают *длинфнс* символов и располагаются в начале строки. Если зна-

чение атрибута *кодблока* есть **истина**, то группы строк могут отделяться кодом конца блока (см. атрибут *кодблока*).

Сообщение текстового а ц д - и п м - ф а й л а содержит некоторое число строк. Строка не может переходить из одного сообщения в следующее. Строка должна заканчиваться байтом «конец строки». Сообщение, вводимое в режиме «с приглашением в виде номера строки», должно содержать одну строку.

Значение атрибута *длинфнс*, отличное от нуля, указывает на то, что вводная строка должна содержать свой номер. Этот номер располагается в строке, начиная с позиции *позфнс* и содержит *длинфнс* символов. Номер может попасть в строку двумя способами:

1. При чтении в режиме «с приглашением в виде номера строки» сама система помещает в строку ее номер.

2. В других режимах необходимо при вводе сообщения с клавиатуры поместить в каждую строку сообщения ее номер.

К о н т е к с т ф а й л а. С текстовым файлом, так же как и с файлом объектного кода, можно связать некоторый справочник (см. атрибут *внешконт*). Такой справочник является корневым справочником внешнего контекста текстового файла. В этом контексте могут находиться объекты, ссылки на которые тем или иным образом обозначены в тексте файла или закодированы в его атрибутах (см. ниже атрибут *имяяз*).

И м я я з ы к а. Содержимое текстового файла может представлять собой текст программы на некотором языке программирования. В этом случае атрибут *имяяз* содержит строку литер (S), представляющую собой имя программы транслятора данного языка программирования. Если предположить, что X — это указатель, установленный на корневой справочник внешнего контекста текстового файла, то ⟨внешнее имя⟩ X//S приводит к нужному транслятору. Одно из использований этого атрибута связано с неявной трансляцией текста, происходящей в том случае, если в конструкции ⟨вызов⟩ вместо процедуры используется текстовый файл (см. § 2.1 этой главы и § 12 гл. 2). ■

13. Изображение файла

В программе файл может появиться следующими способами:

1. Программа может создать новый файл с помощью генератора.

2. Программе, как фактический параметр, может быть передан существующий файл.

3. Программа может обратиться к существующему файлу с помощью ⟨внешнего имени⟩.

4. Файл может быть непосредственно изображен в тексте программы аналогично числовым константам и массивам констант. Такой файл называется статическим файлом. Статический файл изо-

бражается в программе с помощью конструкций «изображение файла».

⟨статический файл⟩ ::=
⟨изображение произвольного файла⟩
| ⟨изображение текстового файла⟩
⟨изображение произвольного файла⟩ ::=
файл ⟨вложенный файл⟩
⟨изображение текстового файла⟩ ::=
тфайл {(⟨список установок атрибутов⟩)}
⟨определение ограничителя⟩ ⟨текст файла⟩ ⟨ограничитель⟩

13.1. Произвольный файл. ⟨Изображение произвольного файла⟩ может присутствовать только в программах, текст которых является пк-файлом (т. е. текст программы расположен на перфокартах или на другом носителе, например, диске, на котором в данном случае он должен быть представлен в виде псевдо-чпк-файла).

⟨Вложенный файл⟩ является самостоятельным пк-файлом, имеющим обычную структуру. Он представляет собой последовательность перфокарт, начинающуюся с начальной метки и кончающуюся конечной меткой. В начальной метке вложенного пк-файла определяются его атрибуты. Значения этих атрибутов могут не совпадать со значениями аналогичных атрибутов охватывающего пк-файла (в частности, в текстовый файл может быть вложен двоичный файл). Перфокарты, находящиеся между начальной и конечной меткой, определяют содержимое изображенного файла.

Внутри текстового ⟨вложенного файла⟩ могут опять встречаться ⟨изображения произвольного файла⟩.

Результатом выполнения ⟨изображения произвольного файла⟩ является постоянный указатель внешнего объекта, установленный на данный чпк-файл. Атрибуты файла совпадают с атрибутами, указанными в начальной метке ⟨вложенного файла⟩.

Обычно текст программы с ⟨изображениями произвольных файлов⟩ образуется следующим образом. Отдельно пробивается колода с основным текстом программы (например, текст задания пакетного режима) и отдельно — колоды ⟨вложенных файлов⟩. Затем последние вкладываются в нужные места основной колоды (см. рис. 8).

13.2. Текстовый файл. Конструкция ⟨изображение текстового файла⟩ предназначена для изображения в программе статических файлов, содержащих произвольную текстовую информацию, в частности, текст Эль-76-программ, текст программ на других языках программирования. Статический текстовый файл может быть изображен и с помощью описанного выше ⟨изображения произвольного файла⟩. Однако имеются следующие отличия:

— текст охватывающей программы не обязательно должен быть расположен на перфокартах;

— не требуется особой пробивки для начальной и конечной метки.

Ограничитель текста. В ⟨определении ограничителя⟩ задается закрывающий ограничитель текста файла. Определение ограничителя отыскивается следующим образом. После служебного слова **тфайл** пропускаются (если они есть) пробелы,

Рис. 8. Программа с вложенными файлами

комментарии и ⟨список установок атрибутов⟩ в круглых скобках. Обозначим литеру, следующую за пропущенными, через X (в приведенном ниже примере это литера *). Тогда ограничителем файла является последовательность литер, расположенная между литерой X и следующим вхождением той же литеры (в приведенном примере последовательность представляет собой два восклицательных знака).

Пример.

```
тфайл
% на следующей строке определен ограничитель !!
```

```
*!!*
этот текст из 3 строк
является содержимым
текстового файла
!!
```

Литера X не может быть пробелом, символами %, ▢ и левой круглой скобкой.

Структура текста. Текст файла представляет собой последовательность строк. Первой строкой этой последовательности является строка, следующая за той, на которой определен закрывающий ограничитель файла, а последней строкой является строка, предшествующая той, на которой найдено вхождение закрывающего ограничителя файла.

Атрибуты файла. Результатом выполнения ⟨изображения текстового файла⟩ является постоянный указатель, установленный на данный текстовый мб- или мд-файл. Полученный файл имеет следующие атрибуты:

— *типфайла* = текст. Если текст написан на некотором языке программирования, то этот язык можно указать, задав в ⟨списке установок атрибутов⟩ атрибут *имяяз*;

— *блоклкф* равен номеру последнего заполненного блока плюс 1. Атрибуты *максдлинблока* и *длинлиста* устанавливаются транслятором.

В ⟨списке установок атрибутов⟩, кроме прочих, можно указать атрибуты *длинфнс*, *кодфнс*, *фнс*, *шагфнс*. Тогда в соответствии со значениями этих атрибутов в начало каждой строки будет помещен ее номер, и в результате образуется нумерованный файл. Если эти атрибуты не указаны в ⟨списке установок атрибутов⟩, то полученный файл является нумерованным файлом с атрибутом *длинфнс*, равным 8.

Если внешний контекст текстового файла явно не задан (см. атрибут *внешконт*), то контекстом файла становится внешний контекст программы, в тексте которой изображен данный файл.

14. Статический справочник

Статический справочник представляет собой оптимизированный вариант обычного архивного справочника. Оптимизация достигается в следующих моментах:

— статический справочник транслируется, а не создается специальными операциями;

— статический справочник хранится в компактной форме в файле объектного кода программы;

— доступ к элементам статического справочника реализуется

более эффективными способами, чем в случае обычных справочников.

⟨статический справочник⟩ ::=
спрконст (⟨список элементов справочника⟩)
⟨элемент справочника⟩ ::= ⟨идентификатор⟩ {@} = ⟨выражение⟩

Результатом выполнения ⟨изображения справочника⟩ является постоянный указатель внешнего объекта, установленный на данный статический справочник. Этот справочник состоит из элементов, перечисленных в ⟨списке элементов справочника⟩. Имя каждого элемента задается компонентой ⟨идентификатор⟩, а значение элемента — компонентой ⟨выражение⟩. В отличие от обычного справочника элемент статического справочника может содержать значения любого типа.

Использование. Статические справочники можно использовать в тех же операциях и конструкциях, что и обычные справочники. Исключение составляют операции создания элемента справочника и записи в элемент справочника, которые в данном случае запрещены.

Особенности выполнения. Существует различие в семантике выполнения операций над обычными и над статическими справочниками. В случае статических справочников семантика операций и конструкций, использующих значение элемента справочника, имеет следующую особенность: значение элемента здесь определяется как результат выполнения соответствующего ⟨выражения⟩; выполнение происходит каждый раз при обращении к содержимому элемента.

Значение элемента справочника может приводить к внешнему объекту. Такой элемент статического справочника эквивалентен элементу обычного справочника, содержащему эквивалентную ссылку. Его можно использовать обычным образом, в частности, — как промежуточное звено при поиске по внешнему имени.

Для того чтобы получить непосредственно значение элемента статического справочника, надо воспользоваться конструкцией ⟨генератор объекта связи⟩ со спецификацией *прогр* (см. последнее присваивание в приведенном ниже примере).

Пример.

```
начало
  перем локсправ, файл1, прог1;
  процедура p = проц (. . .) (. . .);
    . . .
    локсправ := спрконст (
      x1 = //спр1,
      x2 = p);
```

```
...
файл1 := файл (локсправ//х1 //фл1);
прог1 := прогр (локсправ // х2);
...
конец
```

Здесь предполагается, что во внешнем контексте программы есть справочник//*спр*1, в который под именем "*фл*1" занесена ссылка на некоторый файл. Тогда с точки зрения выполнения второго оператора элемент "*х*1" эквивалентен элементу обычного справочника, содержащего ссылку на справочник //*спр*1. Изображенное ⟨внешнее имя⟩ приводит к файлу, который открывается при выполнении этого оператора. В результате открытия программы в правой части третьего оператора выдается процедура *р*.

Косвенные элементы. Присутствие символа @ в ⟨элементе справочника⟩ означает, что содержимое элемента рассматривается как косвенная ссылка. При поиске по контексту подобные элементы играют такую же роль, как и элементы обычных справочников, содержащие косвенные ссылки.

15. Пакет заданий

Пакет заданий представляет собой колоду перфокарт, составленную из последовательности заданий. Задание — это пк-файл, имеющий структуру, изображенную на рис. 9.

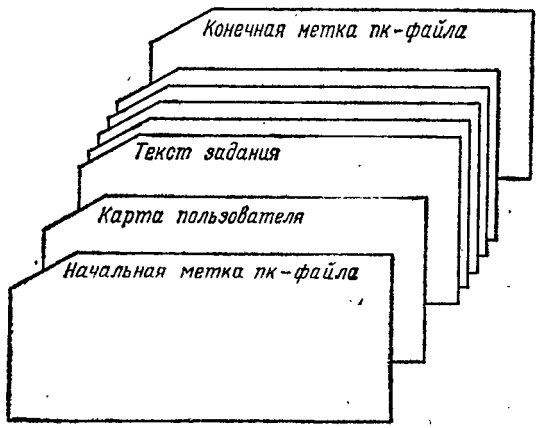


Рис. 9. Колода пакетного задания

Карта пользователя — это перфокарта с именем пользователя, паролем пользователя и прочими атрибутами, позволяющими системе идентифицировать пользователя и найти его внешний контекст. Текст задания — это конструкция ⟨закрытый оператор⟩

или ⟨запуск задачи⟩, в ⟨тексте программы⟩ которой описано содержание задания.

Выполнение пакета состоит в том, что задания вводятся одно за другим и в параллельном режиме инициируется их выполнение.

Выполнение задания состоит в выполнении конструкции ⟨запуск задачи⟩ или ⟨закрытого оператора⟩. Внешним контекстом, в котором происходит выполнение, является контекст, идентифицируемый картой пользователя. Пример задания представлен на рис. 10. На карте 3 задается имя задачи "*тест*". Это имя печатается

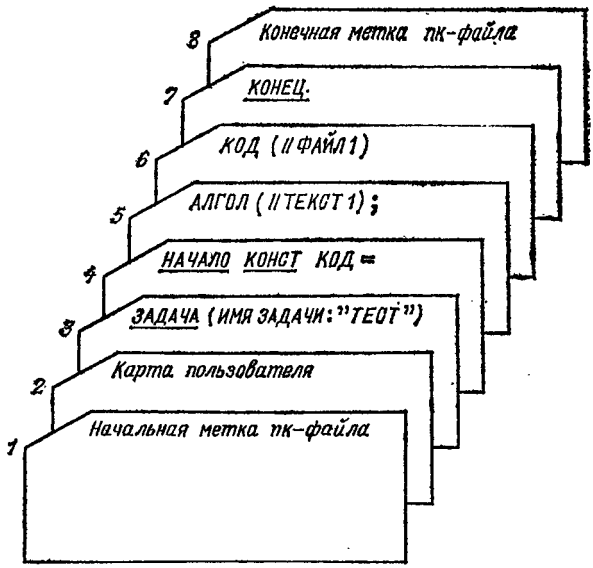


Рис. 10. Пример задания

при выдаче итогов решения задачи. На карте 5 задается трансляция алгол-транслятором текстового файла //*текст*1 (предполагается, что идентификатор *алгол* описан в стандартном контексте и обозначает процедуру, являющуюся алгол-транслятором). Указатель внешнего объекта, установленный на файл объектного кода, присваивается константе *код*. На карте 6 файл объектного кода неявно открывается, и программа выполняется. В качестве фактического параметра передается файл //*файл*1.

16. Ошибки взаимодействия с внешними объектами

В процессе обмена и при работе с архивом могут возникнуть особые обстоятельства, объединяемые под общим названием «ошибки взаимодействия с внешними объектами». Для каждой подобной ошибки система предусматривает стандартную реакцию.

16.1. Номенклатура ошибок. Ниже перечисляются ошибки взаимодействия с внешними объектами и описываются стандартные реакции на каждую ошибку.

1) Ошибка «логический конец файла» возникает в следующих случаях:

— при попытке чтения информации, находящейся за пределами заполненной части файла (это не распространяется на чтение информации из мб-, мд-файла при непосредственном обмене, так как в этом случае система не отслеживает позицию логического конца файла);

— при попытке установить текущую позицию (с помощью операции позиционирования при непосредственном обмене) за пределы заполненной части файла.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситлкф*.

Пользователь может соответствующими средствами языка определить собственную реакцию.

2) Ошибка «физический конец файла» возникает при попытке увеличения объема файла до размера, превышающего максимально возможный.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситфкф*.

Пользователь может соответствующими средствами языка определить собственную реакцию.

3) Ошибка «нет листа» возникает при попытке чтения информации с математического листа мб-, мд-файла, для которого не распределен физический лист.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситнетлиста*.

4) «Ошибка символа» возникает в случае, если значение атрибута *двоичный* для чпк-файла есть **ложь**, и при чтении очередной перфокарты этого файла обнаружено, что какая-либо колонка содержит конфигурацию пробивок, не имеющую эквивалента в коде ДКОИ-8.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситошсмв*.

5) В случае сбоя в работе внешнего устройства система предпринимает ряд мер для перезапуска операции, во время выполнения которой произошел сбой. Если попытки перезапуска не заканчиваются удачей, то возникает «ошибка внешнего устройства».

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситошву*.

6) «Ошибка в параметрах обмена» возникает в том случае, когда нарушен интерфейс с системной процедурой, т. е. параметры

операции обмена по типу, по числовой величине или по их количеству не соответствуют требованиям.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситошпарамобмена.*

7) Ошибка «нет внешнего объекта» возникает при попытке доступа к ликвидированному внешнему объекту.

Стандартная реакция: выполняется структурный переход без параметров по динамической ситуации *ситнетвнеш.*

8) Ошибка «нет ресурса» возникает в случае, если в системе нет ресурсов, чтобы удовлетворить заявку на создание внешнего объекта.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситнетрес.*

9) Ошибка «привилегированный обмен» возникает в случае, если в непривилегированном режиме выполняются привилегированные операции с объектом.

Стандартная реакция: в случае, если в непривилегированном режиме производится обмен данными типа адресной информации, последние преобразуются к типу набор без изменения информационной части. В остальных случаях выполняется структурный переход по динамической ситуации *ситпривобмен.*

10) Ошибка «нет в архиве» возникает в следующих случаях:

— когда в справочнике (или контексте) не найден элемент с данным именем;

— если при позиционировании справочника обнаружено начало или конец справочника.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситнетварх.*

11) Ошибка «нет архивной ссылки» возникает в случае, когда архивная операция, требующая непустой ссылки на объект, встречает пустую ссылку на внешний объект.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситнетархслк.*

12) «Ошибка в параметрах архивной операции» возникает в случае, когда нарушен интерфейс с архивными операциями.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситошпарамарх.*

13) «Ошибка атрибута» возникает при неверном задании атрибута открытия или атрибута внешнего объекта.

Стандартная реакция: выполняется структурный переход по ситуации *ситошатр.*

Ошибки обмена подразделяются на логические и реальные. Логические ошибки обнаруживаются системными процедурами в процессе формирования заявки на обмен. Реальные ошибки обнаруживаются внешними устройствами уже в процессе непосредствен-

ного выполнения операции. Ошибки обмена и их характер указаны в табл. 2.

Таблица 2

Ошибка	1	2
Логический конец файла	р	*ситлкф*
Физический конец файла	р	*ситфкф*
Нет листа	л	*ситнетлиста*
Ошибка символа	р	*ситошсмв*
Ошибка устройства	р	*ситошву*
Ошибка в параметрах обмена	л	*ситошпарамобмена*
Нет внешнего объекта		*ситнетвнеш*
Нет ресурса		*ситнетрес*
Привилегированный обмен	р	*ситпривобмен*
Нет в архиве		*ситнетварх*
Нет архивной ссылки		*ситнетархслк*
Ошибка в параметрах архивной операции		*ситошпарамарх*
Ошибка атрибута		*ситошатр*

В колонках табл. 2 приведена следующая информация: 1 — характер ошибки обмена (л — логическая ошибка, р — реальная), 2 — идентификатор стандартной динамической ситуации на ошибку.

Стандартная реакция аварийного характера состоит в следующем. Если в процессе выполнения какой-либо процедуры синхронного обмена (буферизованного или непосредственного) обнаружена логическая или реальная ошибка или же если при выполнении процедуры запуска параллельного обмена обнаружена логическая ошибка, то это приводит к прекращению выполнения процедуры обмена в результате структурного перехода по стандартной динамической ситуации, соответствующей обнаруженной ошибке. Программа может определить реакцию на возникновение ситуации. Для этого нужно, чтобы место вызова процедуры синхронного обмена охватывалось (в общем случае динамически) структурным предложением, управляющим этой ситуацией. Например:

```
до * ситлкф
    (чит (ф1, след, м1) )
при
    ситлкф : запф (печ,: "конец файла")
всесит
```

Если же такого динамически охватывающего структурного предложения нет, то выполнение независимого процесса прекращается и предпринимается конечная реакция на ситуацию, предусмотренная системой.

Если в процессе выполнения параллельного обмена обнаружена реальная ошибка, то структурный переход по соответствующей ситуации происходит не из места вызова процедуры запуска параллельного обмена, а из места, где выполняется операция синхронизации задачи с процессом обмена. Например, пусть в программе есть два ⟨структурных предложения⟩:

```
до * ситлкф
    (читпар (ббв1, след, м1))
при
    ситлкф:. . . R1 . . .
всесит;
    . . .
до * ситлкф
    (ждать (имя читатр (ббв1, смфобмена) @))
при
      ситлкф:. . . R2. . .
всесит
```

При возникновении ошибки «логический конец файла» будет выполняться действие R2, а не R1. ■

ГЛАВА 4

АТРИБУТЫ ОБЪЕКТОВ И АТРИБУТЫ ЗАДАЧИ

В этой главе описываются основные атрибуты внешних объектов, оперативных объектов связи и буферов, атрибуты паспорта в-массива и атрибуты задачи. Описанию семантики атрибута предшествует следующая информация:

— идентификатор атрибута;

— тип внешнего устройства, на котором может располагаться внешний объект;

— принадлежность атрибута. Здесь используются следующие условные обозначения: О — атрибут открытия, В — атрибут внешнего объекта;

— перечень конструкций, в которых можно использовать атрибут. Здесь используются следующие условные обозначения: Г — атрибут можно указывать в генераторе объекта (для атрибутов открытия — в ⟨генераторе объекта связи⟩, а для атрибутов внешнего объекта — в ⟨генераторе внешнего объекта⟩ или в ⟨генераторе объекта связи⟩ для случая неявной генерации внешнего объекта), М — возможна модификация атрибута. ГВ — для атрибутов открытия обозначает, что умолчание для этого атрибута можно хранить в заголовке внешнего объекта. Если использование атрибута в некоторой конструкции разрешено не для всех типов внешних устройств, то те типы, для которых оно разрешено, перечислены в скобках после обозначения данной конструкции;

— в квадратных скобках указан тип значения атрибута. Если атрибут можно задавать в генераторе, то следом указывается значение, которым атрибут инициализируется по умолчанию. Тип обозначается следующим образом: Б — **истина** или **ложь**, Ц — целое, С — байтовый вектор или набор, У — объект связи или с-вектор, М — процедура.

1. Общие атрибуты оперативных объектов

типоб
О; [Ц]

Значение атрибута определяет тип оперативного объекта. Атрибут может принимать следующие значения:

1 — заголовок открытого контейнера;
2 — заголовок открытого файла;
3 — позиционная переменная;
4 — таблица буферов;
5 — блок ввода/вывода;
8 — подвижный указатель внешнего объекта.

локал
О/Г; [Б или *ствж*, **истина**]

Значение атрибута определяет время жизни объекта. Атрибут может принимать следующие значения:

истина — объект является локальным;
ложь — объект является глобальным;
объект типа *ствж* — время жизни объекта, такое же, как время жизни объекта, являющегося значением атрибута.

типвнешоб
О; [Ц]

Значение атрибута определяет разновидность внешнего объекта, на который установлен указатель внешнего объекта.

1 — контейнер,
2 — файл,
4 — объект контекста съемных носителей,
5 — справочник глобального архива,
10 — справочник.

2. Атрибуты файла

2.1. Характеристики внешнего устройства.

типву
любой; В/Г; [Ц]

Значение атрибута определяет тип внешнего устройства. Возможны следующие значения:

1 — магнитный барабан (мб);
2 — магнитный диск (мд);
3 — магнитная лента (мл);
4 — ввод с перфокарт (чпк);

5 — вывод на перфокарты (зпк);
6 — ввод с перфоленты (чпл);
7 — вывод на перфоленту (зпл);
8 — пишущая машинка (пм);
9 — алфавитно-цифровой дисплей;
10 — алфавитно-цифровое печатающее устройство (ацпу).

плотность
мл; В/Г; [Ц]

Атрибут определяет плотность записи на магнитную ленту. Возможны следующие целочисленные значения:
0—8 байтов/мм;
2—32 байта/мм;
3—63 байта/мм.

кодблока
чпл, зпл; О/Г; [Б, **ложь**]

Если значение атрибута — **истина**, то конец блока на перфоленте обозначается специальным символом, набираемым на панели устройства.

двоичный
зпк, чпк; В/Г; [Б, **ложь**]

Если значение атрибута — **истина**, то информация на перфокартах представлена в двоичном коде, в противном случае — в перфокарточном коде КПК-12.

2.2. Характеристики физических границ.

максдлинблока
любой; В (мб, мд, мл, пк, ацпу)/О(пл, ацд, пм)/Г/М;
[Ц, см. семантику]

Значение атрибута определяет максимальное число элементов в блоке файла и тем самым задает размер буфера, используемого при буферизованном обмене. Для мб- и мд-файлов число элементов задается в словах, для файлов остальных типов — в байтах. Умолчание выбирается в зависимости от типа: для мб-, мд- — 256 слов; для пк-файлов, если значение атрибута *двоичный* есть **истина**, то 160 байтов, в противном случае — 80 байтов; для ацпу-файлов — 128 байтов; для пм-, ацд-файлов — в зависимости от размера экрана устройства; для мл-, пл-файлов — 256*8 байтов.

Атрибут можно модифицировать только тогда, когда файл не открыт для буферизованного обмена.

длинлиста
мб, мд; В/Г/М; [Ц, 128]

Значение атрибута определяет количество секторов в физическом листе памяти мб- и мд-файла. После того как для файла отведен первый физический лист, атрибут можно только запрашивать.

максдлинфайла
мб, мд; В/Г/М; [Ц, 10]

Значение атрибута определяет максимально возможный объем файла. Объем мб- и мд-файлов задается в терминах числа листов.

чслфлистов
мб, мд; В/Г; [Ц, 0]

Значение атрибута определяет число физических листов, отведенных в данный момент для файла. Начальное значение атрибута задает число листов, отводимых для файла в момент его создания.

максфлист
мб, мд; В; [Ц]

Значение атрибута определяет максимальный номер математического листа, для которого существует соответствующий ему физический лист.

обрезан
мб, мд; В/Г/М; [Б, ложь]

Атрибут может принимать следующие значения: **ложь** — при откреплении файла система не будет осуществлять сокращение последнего листа файла; **истина** — при откреплении файла система произведет сокращение последнего листа в соответствии с атрибутом *блоклкф* и освободившийся остаток отдаст в область свободной памяти на носителе.

максдлинстроч
ацпу; О/Г/ГВ; [Ц, 128]

Значение атрибута определяет максимальное количество литре на строчке бумажного носителя ацпу-файла.

максдлинстран
ацпу; О/Г/М/ГВ; [Ц, 72]

Если значение атрибута отлично от нуля, то оно определяет максимальное количество строчек, располагающихся на логической странице бумажного носителя ацпу-файла.

2.3. Характеристики логических границ.

блоклкф
мб, мд; В/М; [Ц]

Значение атрибута определяет номер первого свободного блока файла.

элемлкф
мб, мд; В/М; [Ц, 0]

Значение атрибута задает позицию логического конца файла в последнем заполненном блоке.

длинблока
мл, чпл, ацд, пм; О; [Ц]

Значение атрибута определяет реальную длину текущего блока файла (в байтах).

2.4. Характеристики текущей позиции.

блокфайла
мл, пк, ацпу, пл; О; [Ц]

Значение атрибута определяет порядковый номер текущего блока (относительно начала файла), в конец которого реально установлена головка записи/считывания.

строчка
ацпу, ацд; О/Г/М(ацд); [Ц]

Если значение атрибута *максдлинстран* отлично от нуля, то значение этого атрибута определяет номер текущей строчки. Номер строчки отсчитывается от начала текущей логической страницы (номер первой строчки равен 0).

страница
ацпу; О/Г/М; [Ц]

Если значение атрибута *максдлинстран* отлично от нуля, то значение этого атрибута определяет номер текущей логической страницы.

Номер текущей страницы увеличивается на 1, когда номер строчки становится равным значению атрибута *максдлинстран*. При этом номер строчки обнуляется.

2.5. Атрибуты, связанные с именованием.

имяфайла
мб, мд, мл, пк, ацпу, пл; В (мб, мд, мл, пк, ацпу)/Г(мб, мд, мл, пк, ацпу);
[С, пустое имя (пк, пл, ацпу)]

Значением атрибута является указатель байтовой строки или байтовый набор. В обоих случаях последовательность символов — это имя из метки или заголовка файла. Кроме того, есть непомеченные файлы. Непомеченный файл не имеет метки, где хранились бы

его атрибуты. Поэтому атрибуты непомеченного файла являются атрибутами открытия. Стратегия постановки на устройство нужного непомеченного файла определяется оператором. Пл-файл всегда является непомеченным.

Через посредство указателя *нпо* можно обрабатывать помеченный мл-файл и мл-контейнер как непомеченные.

2.6. Даты.

датасозд
мд, мб, мл; В/М; [С]

Значение атрибута определяет дату создания объекта в формате *ггммдд*, где *гг* — две последние цифры года, мм — номер месяца, а *дд* — номер дня в месяце.

послдата
мб, мд; В/М; [С]

Значение атрибута определяет дату последней модификации файла (в формате *ггммдд*).

2.7. Характеристики распределения листов.

файлвпакет
мб, мд; В/Г; [Б, **ложь**]

Если значение атрибута — **истина**, то физические листы данного файла располагаются на одном томе.

листвцилиндр
мд, мб; В/Г; [Б, **ложь**]

Если значение атрибута — **истина**, то каждый физический лист файла полностью располагается на одном цилиндре. Для мб-файлов значение атрибута всегда равно **истина**.

порядоклистов
мд, мб; В/Г; [Ц, 0]

Если значение атрибута равно числу 1, то физические листы файла располагаются в порядке возрастания номеров соответствующих математических листов. Если значение атрибута равно числу 2, то листы располагаются в порядке убывания. Если значение атрибута равно нулю, то листы располагаются в произвольном порядке.

урп
мд, мб; В/Г; [Б, **ложь**]

Атрибут задает режим управляемого программистом распределения памяти на носителе. Это означает, что программист с

помощью атрибута *базалиста* должен указывать начальные адреса каждого листа файла.

Режим управляемого распределения памяти можно задавать только в привилегированном режиме.

2.8. Характеристики текстовых файлов.

длинстрок
любой; В/Г; [Ц, 0]

Если значение атрибута отлично от нуля, то файл состоит из строк одинаковой длины, равной этому значению. В противном случае файл состоит из строк переменной длины. В обоих случаях длина строк не должна превышать значения атрибута *максдлинблока*.

пакстрок
любой; В/Г (мб, мд, мл); [Б, **истина** (мб, мд, мл)/**ложь** (пк, пл, ацпу, ацд, пм)]

Если значение атрибута — **истина**, то пробелы внутри строк упаковываются программой, создающей такой текстовый файл. Это означает, что в состав строк могут входить байты, кодирующие некоторое число подряд расположенных пробелов. Числовая величина таких байтов заключена в пределах $1 \leqslant к \leqslant 63$, где $к$ — количество представленных таким способом пробелов.

Если значение этого атрибута — **ложь**, то упаковка пробелов не производится.

длинфнс
любой; В/Г/М(ацд, пм); [Ц, 0]

Если значение атрибута отлично от нуля, то файл является нумерованным, т. е. каждая строка содержит собственный номер, состоящий из *длинфнс* цифр. В противном случае в файле плавающая нумерация строк, т. е. строка не содержит своего номера.

кодфнс
любой; В/Г(мб, мд, мл); [Ц, 0(мб, мд, мл)/1(пк, пл, ацпу, ацд, пм)]

Для нумерованного файла значение этого атрибута определяет кодировку номера строки. Если значение равно нулю, то кодировка двоичная. В этом случае для номера строки отводится столько байтов, сколько нужно, чтобы уместить двоичный код максимального целого, состоящего из *длинфнс* цифр. Например, при значении *длинфнс*, равном числу 8, максимальное целое занимает 27 битов, и для номера строки отводится 4 байта. Если значение атрибута равно 1, используется символьная кодировка. В этом случае для номера строки отводится *длинфнс* байтов, и номер кодируется в виде последовательности литер, соответствующих цифрам номера.

позфнв
любой; В/Г (зпк, ацпу)/М (ацд); [Ц, 0]

Для нумерованного файла значение этого атрибута определяет номер позиции в строке, начиная с которой располагается номер строки.

фнс
ацд, пм; О/М; [Ц, 0]

Для нумерованного файла значение этого атрибута равно фиксированному номеру текущей строки.

шагфнс
ацд, пм; О/М; [Ц, 1]

Для нумерованного файла значение этого атрибута задает шаг нумерации строк.

кодпрогр
мб, мд; В/М; [Ц, 2]

Значением атрибута является целое число, определяющее номер того элемента СВС (справочника внешних связей) текстового файла, который содержит ссылку на файл объектного кода, полученного при трансляции данного текста.

При модификации этого атрибута значением типа указатель внешнего объекта ссылка, содержащаяся в указателе, помещается не в атрибут, а в соответствующий элемент СВС.

имяяз
мб, мд; В/М [С, пустая строка]

Содержимое атрибута — это строка литер, задающая имя программы транслятора языка программирования, на котором написан данный текст. Имя ищется в контексте данного файла (см. атрибут *внешконт*). Если значение атрибута — пустой набор или пустая строка, то содержимое файла — это произвольный текст.

2.9. Прочие характеристики файла.

типфайла
любой; В/Г/М; [Ц, 0]

Значение атрибута характеризует структуру информации, составляющей содержимое файла:

0 — данные произвольной структуры;
2 — файл объектного кода;
3 — текст.

областьзаг
любой; В(мб, мд)/О (остальные типы)/Г; [Ц/У, 0]

Значение атрибута, указанное при создании файла, определяет размер области пользователя в заголовке файла. Семантика модификации и запроса этого атрибута описана в § 3 гл. 3.

типданных
любой; В/Г; [Б, ложь]

В зависимости от значения атрибута *типданных* информация о типе данных теряется при обмене или сохраняется. Если значение атрибута есть **ложь**, то при записи теги отбрасываются. При чтении приписывается тип «полный набор». Если значение атрибута есть **истина**, то запись и считывание не приводят к потере информации о типе.

Обмен данными типа адресной информации в непривилегированном режиме приводит к возникновению ошибки «привилегированный обмен».

восстановимый
мб, мд; В/Г; [Ц, 0]

Если этот атрибут задан при генерации файла, то система с целью организации восстановления файла при сбое прописывает вновь распределяемые листы файла значением этого атрибута. Значение атрибута может заключаться в пределах от 0 до 255.

классфайла
мб, мд; В/Г/М; [Ц, 0]

Этот атрибут используется для того, чтобы установить одинаковый класс всем листам файла (см. атрибут *класслиста*).

терминал
ацд; О; [Ц]

Значение атрибута определяет тип терминала, соответствующего ацд-файлу: 0 — ЕС 7066, 1 — ЕС 7927, 2 — Видеотон 340, 3 — Видеотон 52100.

форматэкрана
ацд; Г/М; [С]

Атрибут позволяет задавать формат изображения на экране терминала. Значение атрибута является полным набором. Параметры формата кодируются в ВРП набора следующим образом: в разрядах 0 ÷ 31 задается количество символов в строке, а в разрядах 32 ÷ ÷63 — количество строк.

чслкопий
мб, мд; В/Г; [Ц, 1]

Значение атрибута задает количество дублей файла, которое автоматически создается системой.

длинсектора
мб, мд; В/Г; [Ц, 32]

Значение атрибута задает физический объем сектора мб-, мд-файла (в терминах количества слов).

наччслбуферов
любой; О/Г/М/ГВ; [Ц, 2]

Значение атрибута определяет, какое число буферов создается для файла в момент его первого открытия для буферизованного обмена в рамках каждой задачи.

Атрибут можно модифицировать только тогда, когда файл не открыт для буферизованного обмена.

запасбуферов
мб, мд; О/Г/ГВ/М; [Ц, 0]

Значение атрибута определяет число резервных буферов на каждое открытие данного файла.

Атрибут можно модифицировать только тогда, когда файл не открыт для буферизованного обмена.

приглашение
ацд, пм; О/Г/М; [Ц/С, 0]

Значение атрибута задает режим диалога, осуществляемого с помощью терминального файла. Нулевое значение атрибута задает режим «без приглашения». В этом режиме операция чтения работает обычным образом. Значение атрибута, равное 1, задает режим «с приглашением в виде номера строки». В этом режиме операция чтения сначала осуществляет приращение текущего значения атрибута *фнс* на значение атрибута *шагфнс* и выводит на носитель номер новой строки в виде *длинфнс* литер. Затем осуществляет чтение ответного сообщения. Если производилась модификация атрибута *фнс*, то первая из последующих операций чтения не осуществляет приращение вновь установленного значения.

Если значением атрибута является строка, осуществляется режим «с приглашением» в виде строки. В этом режиме приглашение выводится в виде строки, являющейся значением атрибута.

активность
ацд, пм; О; [М]

Если значением атрибута является процедура, то эта процедура запускается при возникновении независимой активности со стороны терминала.

времяобмена
любой; О; [Ц]

Значение атрибута определяет суммарное время, затраченное на обмен с данным файлом.

длинсво
мб, мд; В/Г/М; [Ц, 0]

Значение атрибута задает количество элементов в справочнике внешних связей файла. Значение атрибута можно изменять в сторону увеличения.

внешконт
мб, мд; В/Г/М; [Ц, 1]

Значением атрибута является целое число, определяющее номер элемента в справочнике внешних связей файла объектного кода, содержащего ссылку на внешний контекст программы.

При модификации этого атрибута значением типа указатель внешнего объекта ссылка, содержащаяся в указателе, помещается в соответствующий элемент справочника внешних связей.

Запись целого обычно делает транслятор, а пользователь затем записывает ссылку на внешний контекст.

В случае, если текст (или файл с данными произвольной структуры) содержит внешние имена файлов, к которым необходимо обращаться в процессе чтения текста, то этот же атрибут можно использовать и для ссылки из текстового файла (см. также атрибут *имяяз* текстового файла).

текстпрогр
мб, мд; В/М; [Ц, 3]

Атрибут определяет номер того элемента справочника внешних связей файла объектного кода, который содержит ссылку на файл исходного текста программы. В остальном семантика аналогична атрибуту *внешконт*.

инфпрогр
мб, мд; В/М; [Ц, 0]

Атрибут определяет номер того элемента справочника внешних связей файла объектного кода (ФОК), который содержит ссылку на расширение ФОК'а (локальные словари программы, справочник локальных словарей и прочая информация, используемая системой при выдаче сообщений об аварийном прекращении выполнения программы, для символьной отладки программ, при комплексации независимо транслированных программ и т. д.). В остальном семантика аналогична атрибуту *внешконт*.

имяпол
В; [С]

Значением атрибута является байтовая строка, содержащая имя пользователя, создавшего данный файл.

переполнен
мл, чпл, ацд, пм; О; [Ц]

Если после считывания очередного блока файла оказывается, что его реальная длина больше, чем длина массива, в который производится считывание, то значением этого атрибута становится 1, если длины равны, то значением атрибута становится 0, в противном случае значением атрибута становится —1.

запконтроль
мб, мд; О/М; [Б, **ложь**]

Если значение атрибута — **истина**, то запись на носитель осуществляется с последующим контрольным считыванием.

3. Атрибуты листа

класслиста
мб, мд; В; [Ц, 0]

Атрибут *класслиста* определяет том контейнера, на котором размещается данный лист.

Если отводится участок памяти для листа нулевого класса, то занимается память на любом томе, на котором есть свободный участок нужной длины. Для листа ненулевого класса отводится память на томе, класс которого равен классу листа. Если таких томов нет (или есть, но на них нет свободной памяти нужной длины), то отводится память на каком-либо томе нулевого класса, на котором есть свободная память нужной длины. Этому тому приписывается класс листа. Тому возвращается нулевой класс после того, как освобождаются все его участки, отданные под листы с ненулевым классом. Таким образом, данный атрибут позволяет группировать физические листы файла на одном устройстве или на группе устройств с одинаковым классом. Класс должен быть приписан математическому листу до того, как распределяется память для соответствующего физического листа. После того, как память для физического листа отведена, изменять класс данного листа нельзя.

распределен
мб, мд; В/М; [Б, **ложь**]

Если значение атрибута — **истина**, то это означает, что для данного математического листа распределен соответствующий фи-

зический лист. Отведение физического листа, соответствующего некоторому математическому, осуществляется либо неявно (в момент первой записи в этот лист), либо явным образом, с помощью модификации атрибута *распределен*. Если значением атрибута *распределен* становится **ложь**, то освобождается участок, занятый соответствующим физическим листом. Освободившийся участок поступает в область свободной памяти на носителе. Если файл используется в буферизованном обмене, то ликвидируются все буфера, соответствующие блокам, расположенным на освобождаемом листе.

базалиста
мб, мд; В/М; [Ц]

Значение атрибута определяет адрес начала листа на носителе (номер математического пакета плюс физический адрес начала листа внутри пакета). В непривилегированном режиме можно только запрашивать значение этого атрибута. В привилегированном режиме, при значении атрибута *урп* — **истина**, можно устанавливать и изменять значение этого атрибута.

4. Атрибуты контейнера

4.1. Характеристики внешнего устройства.

типву
мб, мд, мл; В; [Ц]

Значение атрибута определяет тип внешнего устройства, на котором располагается контейнер. Атрибут может иметь следующие значения: 1 — магнитный барабан; 2 — магнитный диск; 3 — магнитная лента.

4.2. Характеристики физических границ.

чслфпак
мб, мд; В/Г; [Ц, 1]

Значение атрибута определяет число физических томов, отведенных для контейнера в данный момент. Начальное значение этого атрибута задает число томов, отводимых для контейнера в момент его создания.

максфпак
мб, мд; В; [Ц]

Значение атрибута определяет максимальный номер математического тома, для которого существует соответствующий ему физический том.

4.3. Характеристики текущей позиции.

позконт
мл; О/М; [С]

Атрибут может принимать значения типа набор: "*начк*" — начало контейнера, "*конк*" — конец контейнера, "*начф*" — начало файла, "*конф*" — конец файла.

файлконт
мл; О/М; [Ц]

Значение атрибута определяет порядковый номер текущего файла относительно начала контейнера. При модификации атрибута текущая позиция перемещается на файл с заданным номером.

направление
мл; О/М; [Б, **истина**]

Атрибут определяет позицию, в которую перематываются файлы контейнера при закрытии. При прямом направлении (**истина**) файл сматывается на конец, а при обратном (**ложь**) — на начало.

4.4. Прочие атрибуты.

нмртома
мл; В(мл с меткой)/О(мл без метки)/Г

На конкретной установке машины каждый физический том имеет уникальный регистрационный номер. В случае помеченных мл-томов регистрационный номер назначается при разметке тома, наносится на его внешней поверхности и записывается в метку тома. Атрибут *нмртома* задает регистрационный номер мл-контейнера. При генерации и открытии контейнера с помощью этого атрибута можно управлять выбором конкретного физического тома.

имяконт
мб, мд, мл; В(мб, мд, мл с меткой)/О (мл без метки)/Г/М (мд); [С]

Значением атрибута является имя, содержащееся в метке базового тома мб- или мд-контейнера, в метке помеченного мл-контейнера, или имя, используемое оператором системы для выбора нужного непомеченного мл-контейнера.

свобпам
мб, мд; О; [У]

Значением атрибута является указатель внешнего объекта, установленный на контейнер. Особенность этого указателя состоит в том, что он не обеспечивает возможности доступа к заголовку и

базовому справочнику контейнера, но с его помощью можно создавать объекты в данном контейнере.

имяпол
мб, мд, мл; В(мб, мд, мл с меткой)/О(мл без метки); [С]

Значением атрибута является указатель байтовой строки, содержащей имя пользователя, создавшего данный контейнер.

5. Атрибуты тома

распределен
мб, мд; В/М; [Б, ложь]
Если значение атрибута истина, то для данного математического тома распределен некоторый физический том.

упп
мб, мд; В; [Ц, 0]

При разметке тома можно часть памяти тома квалифицировать как *упп*-область, т. е. область, предназначенную для явного распределения памяти, отводимой для листов файлов, имеющих значение атрибута *упп* — истина. Значение атрибута *упп* тома равно размеру этой области. Адреса начала листов файлов указываются с помощью атрибута *базалиста*. Адреса надо задавать так, чтобы листы размещались внутри *упп*-области.

6. Атрибуты буфера

блокфайла
любой; О; [Ц]

Значение атрибута определяет номер блока, которому соответствует буфер, или обозначает позиции начала или конца файла (см. п. 9.2 гл. 3): —1 — позиция «начало файла»; *блкф* — позиция «конец файла». В двух последних случаях реальный буферный массив не создается.

длинблока
любой; О/М; [Ц]

Значение атрибута определяет длину блока, считанного в буфер (при чтении), или длину части буферного массива, записываемой в файл.

модифицирован
любой; О/М; [Б]

Значение атрибута определяет, выполнялась ли операция записи в буферный массив (зап, нов).

переполнен
любой; О; [Ц]

Если в результате считывания блока в буфер оказывается, что реальная длина блока больше, чем значение атрибута *максдлинблока*, то значением этого атрибута становится 1, если длины равны, то значением атрибута становится 0, в противном случае значением атрибута становится —1.

смфобмена
любой; О; [У]

Значением атрибута является указатель семафора завершения операции обмена с данным буфером.

7. Атрибуты позиционной переменной

С помощью позиционной переменной можно запрашивать все атрибуты текущего буфера (см. в § 3 гл. 3). Кроме того, существует ряд атрибутов открытия, хранящихся непосредственно в позиционной переменной. Эти атрибуты перечислены ниже.

формэлем
любой; О/Г/М/ГВ(мб, мд); [Ц, 6(мб, мд)/3(мл, пк, ацпу, пл, ацд)]

Значение атрибута определяет формат элемента буферного массива. Атрибут может принимать следующие значения: 0 — ф1; 2 — ф4; 3 — ф8; 5 — ф32; 6 — ф64; 7 — ф128.

элемблока
любой; О/М; [Ц]

С помощью этого атрибута пользователь может указывать номер текущего элемента блока.

прикрепфайл
любой; О; [У]

Значением атрибута является заголовок открытого файла, прикрепленного к данной позиционной переменной.

областьпозп
любой; О/Г; [Ц/У, 0]

Значение атрибута, указанное при создании позиционной переменной, определяет в словах размер области пользователя в

позиционной переменной. В этой области пользователь может хранить свою информацию. При запросе этого атрибута выдается указатель, описывающий область.

8. Атрибуты блока ввода/вывода

смфобмена
любой; О/М; [У]

Значением атрибута является указатель семафора завершения параллельного обмена с использованием данного БВВ.

вобмене
любой; О; [Б]

Если значение атрибута — **истина**, то операция обмена с участием данного БВВ еще не окончена.

прервцп
любой; О/Г/М; [Б, **ложь**]

Если значение атрибута — **истина**, то по завершении очередного обмена происходит прерывание центрального процессора. Атрибут можно задавать только в привилегированном режиме.

9. Атрибуты таблицы буферов

При многобуферном способе буферизованного обмена атрибуты буфера обрабатываются с использованием (уточнения) (см. § 3 гл. 3). Кроме того, существует ряд атрибутов открытия, хранящихся непосредственно в таблице буферов. Это атрибуты *формэлем* и *прикрепфайл*. Их семантика совпадает с семантикой аналогичных атрибутов позиционной переменной.

10. Атрибуты элемента справочника

Перечисленные ниже атрибуты элемента справочника обрабатываются через посредство указателя внешнего объекта, установленного на элемент справочника.

косвслк
В/М; [Б; **ложь**]

Если значение атрибута — **истина**, то элемент справочника содержит косвенную ссылку.

имяэлспр
В; [С]

Значением атрибута является строка литер, содержащая символьное имя элемента справочника. Запрос атрибута производится, как в случае других строковых атрибутов, с помощью ⟨уточнения⟩. Длина имени не должна превышать 17 символов. Элементы справочника упорядочены по возрастанию числовых величин имен. Отношение числовых величин имен определяется, как в операции ⟨сравнения строк⟩ (см. гл. 2, п. 14.3), причем при сравнении имена, содержащие менее 17 символов, дополняются справа байтовыми элементами, имеющими значение 4"00". В результате сравниваются строки одинаковой длины.

типобслк
В; [Ц]

Значение атрибута определяет разновидность внешнего объекта, ссылка на который содержится в элементе справочника. Атрибут может принимать такие же значения, как атрибут *типвнешоб* указателя внешнего объекта.

адресзаписи
В; [Ц]

Значение атрибута равно адресу объекта, ссылка на который содержится в элементе справочника.

11. Атрибуты паспорта

Ниже приводятся атрибуты объекта типа паспорт. Значение атрибута *нигрнуль* можно запрашивать и изменять. Остальные атрибуты можно только запрашивать. Атрибуты *длинизм* и *нигрнуль* определены только для прямоугольного параллелепипеда (см. атрибут *формпар*).

чслизм

Значение атрибута равно числу измерений выстроенного массива.

длинизм

С помощью этого атрибута можно запрашивать длины по измерениям. Для этого в конструкции ⟨запрос атрибута⟩ на месте компоненты ⟨уточнение⟩ надо указать либо номер измерения, например *читатр* (*а*, 1, *длинизм*), либо с-вектор, длина которого равна числу измерений, например

читатр (*а*, *вд*, *длинизм*).

В первом случае значением конструкции является длина заданного измерения, а во втором элементам вектора присваиваются

длины всех измерений; значением конструкции является указатель вектора *вд.*

нигрнуль

Если значение атрибута — **истина**, то нижние границы по всем измерениям равны нулю. В противном случае значение атрибута — **ложь**. С помощью ⟨модификации атрибутов⟩ этому атрибуту можно придавать значение **истина**. После такой установки паспорт будет по-прежнему описывать исходный массив, но с учетом сдвига всех нижних границ в нуль.

формпар

Если массив имеет форму прямоугольного параллелепипеда, то значение этого атрибута — **истина**; в противном случае значение — **ложь**.

базвект

Значением атрибута является указатель базового вектора, т. е. с-вектора, в котором расположен в-массив.

12. Атрибуты задачи

пам
[Ц]

Объем оперативной памяти.

резидпам
[Ц]

Объем резидентной памяти задачи.

времяцп
[Ц]

Текущее значение времени, в течение которого задача занимала центральный процессор.

времяобмена
[Ц]

Текущее значение времени, потраченного на обмен.

размацпу
[Ц]

Текущее значение объема всех выдач из задачи на ацпу.

максматпам
Г; [Ц]

Максимальный объем математической памяти, которую может использовать задача. Этот атрибут и все другие, определяющие предельные значения объема ресурсов, задавать не обязательно. Они предназначены для того, чтобы ограничить возможность неконтролируемых ошибок, приводящих к тому, что программа начинает неограниченно потреблять ресурс, в данном случае — математическую память.

максвремяцп
Г; [Ц]

Максимальное время, в течение которого задача может занимать центральный процессор.

максвремяобмена
Г; [Ц]

Максимальное время, которое может быть затрачено задачей на обмен.

максразмацпу
Г; [Ц]

Максимальный объем всех выдач на ацпу.

имязадачи
Г; [С]

Значением атрибута является указатель, описывающий литерный вектор, или набор. В обоих случаях это идентификация задачи, которая печатается при выводе итогов работы данной задачи.

имяпол
[С]

Значением атрибута является указатель литерной строки, содержащий имя пользователя, создавшего данную задачу.

виртвыв
[У]

Значением атрибута является позиционная переменная ацпу-файла, который может быть использован программами, выполняемыми в рамках данной задачи, как стандартный файл вывода результатов.

приорзадачи
Г; [Ц]

Значением атрибута является число, определяющее приоритет выполнения задачи.

режимзадачи
Г; [Ц]

Значение атрибута определяет режим выполнения задачи: 0 — пакетный режим, 1 — диалоговый.

ГЛАВА 5

СРЕДСТВА ОБРАБОТКИ ВНУТРЕННЕГО ПРЕДСТАВЛЕНИЯ ОБЪЕКТОВ

В этой главе описывается внутреннее (реализационное) представление (ВРП) объектов и средства его обработки.

Объект стандартного или определяемого типа строится из объектов во внутреннем представлении. При этом либо есть прямые аналоги, как, например, в случае чисел, либо внутреннее представление объекта строится из нескольких объектов, возможно, скрытых типов, которые либо предварительно описаны в интерфейсе модуля стандартного контекста, либо локализованы в операционной системе. Доступ к составным и процедурным объектам осуществляется посредством промежуточных объектов, называемых указателями. Например, доступ к с-вектору осуществляется посредством промежуточного объекта типа указатель с-вектора. Указатель хранится в памяти, отведенной для переменной или константы, значением которой является этот вектор. Аналогичным образом представлены другие составные объекты (кроме чисел и наборов) и процедурные объекты. Таким образом, представителем объекта в операциях является указатель, приводящий к объекту. Представителем числа (набора) является само это число (набор). Некоторые из описываемых ниже операций обеспечивают возможность непосредственной обработки представителей объектов. В описании таких операций вместо термина «значение операнда» будет употребляться термин «непосредственное значение операнда».

Теги. Числа, наборы, указатели и другие объекты, используемые для нужд реализации, состоят из информационного содержимого и в н у т р е н н е г о т е г а. Тег используется для реализации типов объектов и контроля типов. Между тегами и стандартными типами не во всех случаях обеспечивается взаимно однозначное соответствие.

1. Обработка указателей

Если значением *x* является объект типа *mx*, доступ к которому осуществляется посредством указателя типа *my*, то результатом проверок типа *провтип* (*x*, *mx*) и *провтип* (*x*, *my*) в обоих случаях является **истина**. Различие состоит в том, что в контексте, где известно полное описание типа *my*, возможна обработка элементов структуры указателя. Например, *x*#*my.p* обозначает элемент *p* указателя, а *x*#*mx.px* обозначает элемент *px* объекта, к которому приводит указатель. Приведем еще два примера. Пусть значение *x* — вектор стандартного или программно определяемого типа, тогда *x*#*стдеск.адресп* обозначает элемент «поле адреса» указателя вектора, а *x* [*к*] обозначает *к*-й элемент этого вектора. Указатель объекта определяемого типа в общем случае имеет тип *статд*. Поэтому, если значением *x* является объект определяемого типа *mx*, то *x*#*статд.p* обозначает элемент *p* внутренней структуры указателя, а элемент *px* объекта обозначается *x.px*, если тип *x* статически известен, или *x*#*mx.px*, если, например, *x* обозначает переменную динамического типа.

2. Преобразование типа массива

В этом разделе описываются возможные способы преобразования типа массива, преобразование переменной в массив и обратно, а также эквивалентность переменных. Преобразование осуществляется без изменения информационного содержимого области памяти, занимаемой элементами массива или переменной, и без изменения типов хранящихся там объектов. В результате, в отличие от неконтролируемого преобразования типов, сохраняется контроль типов обрабатываемых значений. Ошибки, допущенные при преобразовании типов, будут обнаруживаться динамически (*ошибкаимени* и проч.).

Стандартная функция *измтип* (*a*, *mx*) осуществляет преобразование типа *ma* массива *a* к типу *mx*. Типы *ma* и *mx* должны быть типами массивов, имеющих одинаковое число измерений и состоящих из элементов одинакового простого или строчного формата, Допускаются следующие преобразования:

ma — программно определяемый тип массива переменных (констант), *mx* — стандартный тип массива переменных (констант). В случае одномерных массивов *mx* должен быть типом *стстр* или *стсв*, причем допускается преобразование вектора переменных в вектор констант. Это позволяет при необходимости организовать защиту от записи в массив.

ma — стандартный тип массива переменных (констант), *mx* — программно определяемый тип массива переменных (констант).

В случае одномерных массивов *та* должен быть типом *стстр* или *стсв*, причем также допускается преобразование вектора переменных в вектор констант.

та — стандартный тип вектора переменных, *тх* — стандартный тип вектора констант.

Типом ⟨вызова⟩ функции *изтип* является тип *тх*, указанный вторым параметром.

Изменение формата элемента вектора. Конструкция, используемая для обозначения элемента массива, может иметь вид:

⟨первичное⟩ [⟨формат⟩ ⟨выражение⟩]

Такой вид допускается, если значение ⟨первичного⟩ имеет тип *ствект*. При этом в зависимости от заданного ⟨формата⟩ происходит преобразование типа вектора к другому стандартному типу вектора, который имеет указанный ⟨формат⟩ элементов (см. п. 3.6 гл. 2). Если элементы исходного массива были переменными (константами), то выбирается тип вектора с элементами, являющимися переменными (константами).

В рассматриваемом случае конструкция ⟨элемент массива⟩ обозначает элемент вектора преобразованного типа, имеющий индекс, равный значению ⟨выражения⟩.

В конструкции ⟨подмассив⟩ компонента ⟨формат⟩ используется с тем же смыслом, что и в конструкции ⟨элемент массива⟩. Сначала производится преобразование типа с-вектора, а затем выделяется заданная часть с-вектора преобразованного типа.

Преобразование переменной в вектор. Операция @ преобразует переменную динамического типа в с-вектор.

⟨формирование указателя⟩ ::= @⟨переменная⟩

Результирующий с-вектор состоит из одного или нескольких элементов. В случае переменной строчного или простого формата вектор в зависимости от формата переменной имеет тип *свлог*(1), *свциф*(1), *свлит*(1), *свкор*(1), *свдин*(1) или *свдлин*(1) и состоит из одного элемента того же формата. В случае поля переменной или поля вектора, состоящего из *к* элементов строчного формата, результирующий вектор в зависимости от формата элемента поля имеет тип *свлог*(*к*), *свциф*(*к*), *свлит*(*к*).

Преобразование вектора в переменную производится с помощью конструкции ⟨указуемая переменная⟩.

⟨указуемая переменная⟩ ::= ⟨первичное⟩ @

Если значением ⟨первичного⟩ является вектор одного из типов *свкор*(1), *свдин*(1), *свдлин*(1), то ⟨указуемая переменная⟩ обозначает

переменную, являющуюся элементом вектора с индексом 0. Если значение ⟨первичного⟩ имеет тип *свлог* (*к*1), *свциф* (*к*4), *свлит* (*к*8), причем $к1 \leqslant 64$, $к4 \leqslant 16$, $к8 \leqslant 8$, то ⟨указуемая переменная⟩ обозначает поле вектора, состоящее из *к*1, *к*4 или *к*8 элементов соответствующего строчного формата.

Изображение массива констант. ⟨Массив констант⟩ — это изображение с-вектора, сформированного из констант, перечисленных в круглых скобках.

⟨массив констант⟩ ::=
мконст ⟨формат⟩ (⟨список элементов массива констант⟩)
⟨элемент массива констант⟩ ::=
{⟨число повторений⟩} ⟨выражение⟩
|{⟨число повторений⟩} (⟨список элементов массива констант⟩)
⟨число повторений⟩ ::= /⟨целое⟩/

⟨Выражение⟩, задающее значение элемента, может быть выражением статического класса либо ⟨строкой⟩. Во втором случае подразумевается, что в состав массива входит содержимое строки (а не указатель, описывающий строку).

Значением конструкции является вектор, тип которого зависит от заданного ⟨формата⟩: **ф1** — *стрлог* (*к*), **ф4** — *стрциф* (*к*), **ф8** — *стрлит* (*к*), **ф32** — *стркор* (*к*), **ф64** — *стрдин* (*к*), **ф128** — *стрдлин* (*к*), где *к* — количество элементов данного формата. Типом конструкции является этот же статически связанный тип.

Элементы массива в общем случае могут иметь различные форматы, возможно, не совпадающие с заданным ⟨форматом⟩. Массив заполняется в соответствии с форматом значений элементов. Практическим следствием такой упаковки является следующее: последовательность элементов формата ф32 располагается смежно в нужном количестве слов; строки всегда начинаются с нового слова.

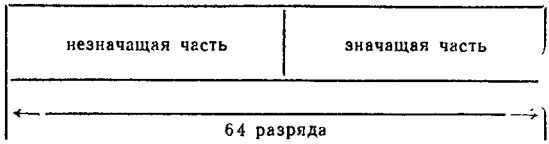
Эквивалентность переменных. В синтаксических правилах п. 5.2, гл. 2 вторая альтернатива правила для ⟨идентификации переменной⟩ предназначена для описания переменной, эквивалентной по адресу некоторой другой ранее описанной переменной. Последняя задается в правой части. Такое описание имеет вид:

⟨простой формат⟩ $x = y$

где x и y — ⟨идентификаторы⟩, причем y также должен быть описан с помощью ⟨описания переменной⟩ с ⟨простым форматом⟩. Формат новой переменной может быть меньше, равен или больше формата эквивалентной переменной. В последнем случае новая переменная накладывается также и на те, которые следуют за эквивалентной переменной.

3. Внутреннее представление наборов

Набор — это объект, состоящий из значащей и незначащей части:



Значащая часть набора типа *наб(к)* содержит *к* элементов формата ф1 и прижата вправо. Под термином «длина набора» понимается длина его значащей части.

Как указано в п. 3.6 гл. 2, набор имеет несколько интерпретаций:

а) Последовательность элементов того или иного строчного формата. В зависимости от операции один и тот же набор можно обрабатывать как битовый, цифровой или байтовый. В описываемых ниже средствах обработки полей элементы набора нумеруются справа налево, начиная от нуля. В конструкции ⟨элемент массива⟩ и ⟨подмассив⟩ элементы нумеруются слева направа. Таким образом, с точки зрения описания типа *наб* как массива, в операциях обработки полей элементы набора нумеруются, начиная от конца массива к его началу (это соответствует использованию наборов в качестве целых, поскольку у чисел разряды обычно нумеруются по степени старшинства справа налево). Если длина набора не кратна подразумеваемому формату элемента, то значащая часть дополняется слева нулями до ближайшей кратной длины. В операциях, требующих типа *лог*, принимается во внимание только правый крайний элемент.

б) Целочисленные операции интерпретируют набор как целое без знака (при этом считается, что значащая часть набора содержит двоичное представление целого).

в) Еще одной областью применения наборов являются вычисления, связанные с обработкой упакованной информации (см. ниже).

4. Упаковка информации в полях

Объект, формат которого не превышает 64 разряда, можно хранить в упакованном и распакованном виде. Таким объектом может являться целое и набор. В упакованном виде объект может содержаться в поле вектора, состоящего из элементов строчного формата, или в поле переменной. Полем вектора называется последовательность нескольких смежных элементов вектора, а полем переменной — часть памяти, занимаемой переменной. Поле вектора может физически начинаться в одном слове памяти и заканчиваться в

другом слове. Переменная и константа строчного формата ф1 (ф4, ф8) обрабатывается как поле, состоящее из одного элемента формата ф1 (ф4, ф8).

В языке существуют средства выборки из поля и присваивания полю. При выполнении присваивания переменной динамического типа, являющейся полем переменной или полем вектора переменных, ВРП непосредственного значения правой части присваивания обрезается слева до длины поля; оставшиеся разряды помещаются в поле. При выборке происходит распаковка; в качестве распакованного представления используется набор, в котором содержимое исходного поля является значащей частью. Таким образом, тип результата выборки не зависит от типа объекта, помещенного в поле. Это, однако, не мешает упаковывать в полях не только наборы, но и целые, так как операции и конструкции, требующие целочисленных операндов, воспринимают набор как целое без знака. Таким образом, после выборки целочисленного значения из поля его не надо преобразовывать к типу целое; это преобразование автоматически делается в операциях.

Средства работы с полями предназначены также для программирования алгоритмов, анализирующих и формирующих объекты во внутреннем представлении.

⟨поле переменной⟩ ::= ⟨переменная⟩.⟨описатель поля⟩
⟨описатель поля⟩ ::=
⟨непосредственный описатель поля⟩
| ⟨идентификатор⟩
⟨непосредственный описатель поля⟩ ::=
[{⟨строчный формат⟩} ⟨выражение⟩ { : {⟨выражение⟩}}]

При определении координат поля считается, что память, занимаемая исходной переменной, состоит из элементов указанного ⟨строчного формата⟩, пронумерованных справа налево, начиная с номера 0. По умолчанию выбирается битовый формат. Местоположение поля определяется по следующей схеме:

где: $к$ — значение п ервого ⟨выражения⟩, т. е. номер начального элемента поля; $к$ — значение второго ⟨выражения⟩, задающего длину поля; $к$ — $к$ + 1 — номер последнего элемента поля.

Местоположение поля либо задается непосредственно, либо указывается с помощью ⟨идентификатора⟩ поля, являющегося синонимом некоторого ⟨непосредственного описателя поля⟩ (см. ниже семантику ⟨описания поля⟩).

Выборка поля. Конструкция ⟨поле значения⟩ осуществляет выборку поля из ВРП непосредственного значения ⟨первичного⟩. В большинстве практических случаев эту и предыдущую конструкцию можно считать синтаксически эквивалентными.

Отличие ⟨поля значения⟩ состоит лишь в том, что (а) оно может быть только элементом ⟨выражения⟩ и (б) его исходный операнд может быть представлен произвольным выражением.

⟨поле значения⟩ ::= ⟨первичное⟩. ⟨описатель поля⟩

Местоположение поля определяется, как указано выше, но вместо переменной рассматривается ВРП непосредственного значения ⟨первичного⟩. ⟨Первичное⟩ может быть динамического типа. Результатом выборки является набор, значащая часть которого представляет собой содержимое заданного поля.

Если в конструкции опущены:

— символ двоеточия и второе ⟨выражение⟩, то считается, что поле состоит из одного элемента;

— второе ⟨выражение⟩, то рассматривается поле от начального элемента до правого края исходного объекта (в случае ⟨поля значения⟩) или до правого края исходной переменной (в случае ⟨поля переменной⟩).

Исходный объект не может иметь тип *двещ*. Значения ⟨выражений⟩ в ⟨описателе поля⟩ должны быть неотрицательными целыми. Если ⟨выражение⟩, задающее длину поля, является выражением статического класса, то ⟨поле значения⟩ имеет статически связанный тип *наб*(*к*), где *к* — значение ⟨выражения⟩. Иначе эта конструкция имеет тип *наб*.

Формирование значения. ⟨Формирование⟩ предназначено для одновременной замены содержимого нескольких полей в ВРП непосредственного значения ⟨первичного⟩. ⟨Первичное⟩ может быть динамического типа.

⟨формирование⟩ ::=
⟨первичное⟩.(⟨список элементов формирования⟩)
⟨элемент формирования⟩ ::=
⟨описатель поля⟩ : ⟨выражение⟩ | **тип** : ⟨выражение⟩

⟨Описатель поля⟩ в ⟨элементе формирования⟩ задает некоторое поле в исходном объекте, куда помещается непосредственное значение ⟨выражения⟩. При этом последнее обрезается слева до длины поля. По умолчанию в качестве исходного объекта выбирается целое число 0.

Результатом выполнения конструкции является исходный объект с модифицированным содержимым указанных полей.

Если в левой части ⟨элемента формирования⟩ находится служебное слово тип, то такой элемент предписывает изменение тега

исходного объекта. В этом случае целое, являющееся значением статического ⟨выражения⟩, задает внутренний тег объекта, полученного в результате ⟨формирования⟩.

Описание поля вводит идентификатор, являющийся синонимом ⟨описателя поля⟩, находящегося в правой части ⟨идентификации поля⟩.

⟨описание поля⟩ ::= **поле** ⟨список идентификации полей⟩
⟨идентификация поля⟩ ::= ⟨идентификатор⟩ = ⟨описатель поля⟩

В данном случае ⟨описатель поля⟩ может содержать только выражения статического класса. Одно ⟨описание поля⟩ может вводить несколько идентификаторов полей.

5. Дополнительные операции

В этом разделе описываются дополнительные операции обработки ВРП объектов, а также дополнительные целочисленные операции.

5.1 Дополнительные одноместные операции.

Знак операции	Операция	Тип операнда	Тип формулы
пче	число единиц	любой	*цел*
перв1	первая единица	любой	*цел*
тип	тип-формат	любой	*наб*(64)
адрес	адрес вектора	с-вектор	*дцел*

Операция **пче** определяет число единиц в ВРП непосредственного значения операнда, а операция **перв1** — номер старшей единицы (если единиц нет, то результат равен —1). Формат значения операнда не должен превышать ф64.

Операция **тип** выделяет внутренний тег непосредственного значения операнда. В поле [7 : 8] результата содержится код тега значения операнда. Если операнд является набором, то в поле [15 : 8] содержится количество битовых элементов в наборе.

Операция **адрес** выдает адрес начала с-вектора.

5.2. Дополнительные операции преобразования типа. Если в перечисленных ниже операциях операнд (*е*) имеет динамический тип, то в операции **целзн** он обрабатывается эквивалентно операнду *е*#*наб*, а в операциях **стнаб** и **млнаб** эквивалентно операнду *е*#*двещ*.

Знак операции	Операция	Тип операнда	Тип формулы
целзн	в целое со знаком	*наб*	дцел
стнаб	удвоенное в набор	*двещ*	*наб*(64)
млнаб	удвоенное в набор	*двещ*	*наб*(64)

Операция **целзн** преобразует полный или неполный битовый набор в целое со знаком формата ф64. Старший бит значащей части набора рассматривается как знак числа и переносится в знаковый разряд.

Операции **стнаб** и **млнаб** преобразуют соответственно старшую и младшую половину ВРП вещественного типа *двещ* в набор типа *наб*(64).

5.3. Дополнительные целочисленные операции. Целочисленные операции +:, —:, *: обеспечивают в случае операндов динамического типа такой же контроль типов значений операндов и результата, как операции +, —, * для операндов целого типа.

Знак операции	Операнд	Тип операндов	Тип формулы
+:	сложение целых	Ц, ДЦ, КЧ, Ч, УНЦ	Ц, ДЦ, Ц, ДЦ, УНЦ
—:	вычитание целых	Ц, ДЦ, КЧ, Ч, УНЦ	Ц, ДЦ, Ц, ДЦ, УНЦ
*:	умножение целых	Ц, ДЦ, КЧ, Ч, УНЦ	Ц, ДЦ, Ц, ДЦ, УНЦ

5.4. Поиск с маскированием. Поиск с маскированием определен для векторов, состоящих из элементов формата ф64. В процессе поиска значение очередного элемента маскируется, т. е. логически умножается на заданную маску, и результат сравнивается с маскированным эталоном. В маскировании участвуют все разряды значения: разряды тега и 64 информационных разряда.

Если заданное отношение не выполняется, производится сравнение следующего элемента. В качестве следующего элемента выбирается следующий элемент вектора или следующий элемент списка в зависимости от того, что предписано операцией. Операция может закончиться после того, как найден нужный элемент, или после исчерпания вектора (алгоритм этой операции отличается от базового алгоритма других групповых операций).

⟨**поиск** по маске⟩ ::=
поиск (⟨выражение⟩, ⟨выражение⟩{, {⟨выражение⟩,}
⟨выражение⟩})
⟨способ выборки⟩ ⟨операция сравнения⟩ ⟨первичное⟩
⟨способ выборки⟩ ::= **до** | **вниздо** | **спискомдо** | **цепьюдо**

Значения первых двух ⟨выражений⟩ задают соответственно вектор и индекс элемента, с которого начинается поиск. Следующие два параметра предназначены для формирования маски. Маска тега задается в правых разрядах ВРП непосредственного значения третьего ⟨выражения⟩, а маска информационной части задается в 64 разрядах ВРП непосредственного значения четвертого ⟨выражения⟩. Эталоном для сравнения является непосредственное значение ⟨пер-

вичного). Последние три значения могут иметь любой тип. По умолчанию маска тега полагается равной нулю, т. е. в процессе поиска тег элементов и тег эталона игнорируются. Маска информационной части по умолчанию заполнена единицами.

Возможны четыре способа выборки следующего элемента.

Служебное слово **до(вниздо)** задает приращение (уменьшение) индекса на каждом шаге на 1. Если поиск прошел успешно, операция выдает индекс найденного элемента. В противном случае признак **тги** устанавливается в **истину**, и выдается индекс, выходящий за границу массива. При прямом направлении обработки он равен длине вектора, а при обратном направлении он равен числу $2^{20}-1$.

Служебные слова **спискомдо** и **цепьюдо** задают поиск по связанному списку. Индекс следующего элемента списка выбирается из поля [19 : 20] текущего элемента. В способе **цепьюдо** ищется либо элемент, для которого выполняется заданное отношение, либо элемент, содержащий «стоп-бит». В обоих случаях первый элемент в сравнении не участвует. Если поиск по списку прошел успешно, то выдается индекс элемента, указывающего на найденный элемент. Индекс имеет отрицательный знак в случае, если при способе выборки **цепьюдо** поиск прекратился на элементе, не удовлетворяющем отношению, но содержащем «стоп-бит». В процессе поиска по списку возможны выход за границу массива и зацикливание. В первом случае возникает ситуация *границамассива*, во втором случае — ситуация *исчерпвремяопрц*.

6. Ввод/вывод внутреннего представления

Конструкция ⟨форматный обмен⟩ обеспечивает возможность ввода и вывода ВРП значений с помощью ⟨трафарета двоичного⟩ и ⟨трафарета тегированного⟩.

Трафарет двоичного. Трафарет двоичного предназначен для ввода/вывода последовательности литер, изображающей внутреннее представление значений.

⟨трафарет двоичного⟩ ::= ⟨основание⟩ *r* ⟨поле цифр⟩ {⟨вставка⟩}
⟨основание⟩ ::= 1 | 3 | 4

Компонента ⟨поле цифр⟩ предназначена для ввода/вывода ВРП значения в двоичном, восьмеричном или шестнадцатеричном виде (в зависимости от ⟨основания⟩). Структура последовательности литер определяется, как и в случае целого без знака. В каждой цифровой позиции находится соответствующая цифра (двоичная, восьмеричная или шестнадцатеричная) ВРП значения. При выводе рассматриваются правые разряды выводимого значения. Число рассматриваемых разрядов равно $(wz + wd)*wr$, где wr равно 1, 3 или 4.

При вводе происходит обратное преобразование: последовательность литер преобразуется в набор (в общем случае — неполный) и присваивается переменной.

При выводе допускаются целые и вещественные числа и наборы. Массив обрабатывается поэлементно, как при выводе чисел. При вводе допускаются переменные и массивы элементов любых форматов.

Количество обрабатываемых разрядов $(wz + wd)*wr$ не должно превышать 64.

Трафарет тегированного. ⟨Трафарет тегированного⟩ предназначен для ввода/вывода последовательности литер, изображающей значение во внутреннем представлении с тегом.

⟨трафарет тегированного⟩ ::= ⟨основание⟩ *t* {⟨вставка⟩}

Последовательность литер имеет форму *ТТ␣УУ. . .У*, где *ТТ* — две шестнадцатеричные цифры 6-разрядного тега значения, *УУ. . .У* — цифры ВРП значения. В зависимости от основания длина этого поля равна 64 (двоичное представление), 22 (восьмеричное представление) или 16 (шестнадцатеричное) цифрам.

Последовательность литер, изображающая вещественное формата ф128, состоит из двух таких последовательностей, разделенных пробелом.

При выводе допускаются значения любого типа. В частности, указатель массива трактуется как выводимое значение.

При вводе допускаются переменные и массивы элементов любых форматов. Вводимая последовательность литер преобразуется в значение, тип которого определяется полем *ТТ* и присваивается переменной или элементу массива. Подобным образом нельзя вводить адресные значения (указатель, метка). В противном случае возникает «ошибка значения».

7. Размещение элементов структуры

С помощью ⟨спецификации размещения⟩ можно управлять распределением памяти для элементов структуры.

⟨спецификация размещения⟩ ::=
[⟨формат⟩ {⟨выражение⟩} : {⟨выражение⟩}]

Значение первого ⟨выражения⟩ задает в терминах ⟨формата⟩ смещение элемента структуры относительно ее начала. Если указано второе ⟨выражение⟩, то оно задает размер области памяти, отводимой для элемента, а иначе этот размер определяется транслятором. Оба ⟨выражения⟩ должны быть статическими выражениями. ⟨Спецификация размещения⟩ либо указывается для всех элементов структуры, либо не используется в описании типа структуры.

ПРИЛОЖЕНИЕ 1

СИНТАКСИС ЯЗЫКА

В этом приложении собраны синтаксические правила языка Эль-76. Каждой группе правил предшествует номер и название параграфа или пункта, в котором вводится эта группа.

Глава 2.

2.2. Лексические элементы.

⟨буква⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N
| O | P | Q | R | S | T | U | V | W | X | Y | Z | У | Ю | Б | Ц | Д | Ф | Г | Й | Л | Я | Ч | Ж | Ы | Ь | З | Ш | Э | Щ

⟨цифра⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨двоичная цифра⟩ ::= 0 | 1
⟨восьмеричная цифра⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
⟨шестнадцатеричная цифра⟩ ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F
⟨литера⟩ ::= ⟨буква⟩ | ⟨цифра⟩ | ⟨вертикальная черта⟩ |
⟨кавычка⟩ | [| . | < | (| + | @ | ^ |] | □ | * | & |) | ; | ‼
| ' |) | # | , | % | > | ! | — | = | ↦ | _ | ⟨пробел⟩
⟨идентификатор⟩ ::= ⟨буква⟩ { ⟨буква или цифра⟩} . . .
⟨буква или цифра⟩ ::= ⟨буква⟩ | ⟨цифра⟩
⟨целое⟩ ::= ⟨цифра⟩ . . .
⟨вещественное⟩ ::= ⟨целое⟩.⟨целое⟩ { ⟨порядок⟩ } | ⟨целое⟩ ⟨порядок⟩
⟨порядок⟩ ::= *e* { ⟨знак⟩ } ⟨целое⟩
⟨знак⟩ ::= + | —
⟨поэлементное представление⟩ ::=
1" ⟨двоичная цифра⟩ . . ." | 3" ⟨восьмеричная цифра⟩. . ."
| 4" ⟨шестнадцатеричная цифра⟩ . . ." | 8" ⟨литера⟩ . . ."
⟨формат⟩ ::= ⟨простой формат⟩ | ⟨строчный формат⟩
⟨простой формат⟩ ::= **ф32** | **ф64** | **ф128**
⟨строчный формат⟩ ::= **ф1** | **ф4** | **ф8**

3.1. Первичные типы.

⟨тип массива⟩ ::= **массив** [{, } . . .] ⟨описание элемента⟩
⟨тип длины⟩ ::= ⟨имя типа⟩ | ⟨отрезок⟩
⟨описание элемента⟩ ::=
конст {#⟨вычисление типа⟩} | **перем** {#⟨вычисление типа⟩}|
конст: ⟨вычисление типа⟩ | ⟨формат⟩

⟨тип области⟩ ::= **область** [⟨тип длины⟩] ⟨вычисление типа⟩
⟨тип структуры⟩ ::=
структ
{ (⟨последовательность параметров размера⟩) }
({ ⟨последовательность описания элементов структуры⟩ })
⟨параметр размера⟩ ::=
⟨список идентификаций параметров длин⟩ |
| **тег** ⟨список идентификаций параметров выбора⟩
⟨идентификация параметра длины⟩ ::=
⟨идентификатор⟩#⟨тип длины⟩
⟨идентификация параметра выбора⟩ ::=
⟨идентификатор⟩#⟨тип выбора⟩
⟨тип выбора⟩ ::= ⟨имя типа⟩ | ⟨отрезок⟩
⟨описание элемента структуры⟩ ::=
⟨описание константы⟩ { ⟨спецификация размещения⟩ }
|⟨описание переменной⟩ { ⟨спецификация размещения⟩}
⟨тип модуля⟩ ::=
интерфейс
{⟨контекст⟩ **начало**}
⟨секция контекста⟩
конинт
⟨тип процедуры⟩ ::=
проц ({⟨последовательность описаний параметров⟩}) |
функция ({⟨последовательность описаний параметров⟩})
{#⟨имя типа⟩}
⟨описание параметра⟩ ::=
{прог} ⟨описание константы⟩
| **{прог}** ⟨описание переменной⟩

3.2. Объединенный тип.

⟨тип объединения⟩ ::=
выбтип ⟨тип выбора⟩ **из**
⟨список составляющих типов⟩
всевыб
⟨составляющий тип⟩ ::= ⟨выражение⟩ : ⟨вычисление типа⟩

3.3. Вычисление типов.

⟨вычисление типа⟩ ::=
⟨изображение типа⟩ | ⟨вычисление подтипа⟩
| ⟨имя типа⟩
⟨изображение типа⟩ ::= ⟨тип массива⟩ | ⟨тип области⟩
| ⟨тип структуры⟩ | ⟨тип модуля⟩
| ⟨тип процедуры⟩ | ⟨тип объединения⟩
⟨вычисление подтипа⟩ ::= ⟨отрезок⟩ | ⟨перечисление⟩ |
⟨элемент области⟩ | ⟨связывание типа⟩ | ⟨выбор типа⟩

⟨имя типа⟩ ::=
⟨идентификатор⟩ | ⟨элемент модуля⟩
| ⟨внешнее имя⟩
⟨отрезок⟩ ::= **до** ⟨выражение⟩
⟨связывание типа⟩ ::=
⟨имя типа⟩ (⟨список выражений⟩)
| **массив** [⟨список выражений⟩] ⟨описание элемента⟩
| ⟨имя типа⟩ (⟨список типов индексов⟩)
| **массив** [⟨список типов индексов⟩] ⟨описание элемента⟩
⟨тип индекса⟩ ::= ⟨имя типа⟩ | ⟨отрезок⟩
⟨выбор типа⟩ ::=
⟨имя типа⟩ (⟨выражение⟩)
| **выбтип** ⟨выражение⟩ **из**
⟨список составляющих типов⟩
всевыб
⟨ограничение областью⟩ ::= **элем** ⟨идентификатор⟩

3.5. Описание типа.

⟨описание типа⟩ ::= **тип** ⟨список идентификаций типов⟩
⟨идентификация типа⟩ ::= ⟨идентификатор⟩ {= ⟨вычисление типа⟩}

3.6. Стандартные типы объектов.

⟨набор⟩ ::= ⟨поэлементное представление⟩ . . . | **истина** | **ложь**|""
⟨число⟩ ::=
{⟨тег целого⟩} ⟨целое⟩
| {⟨тег вещественного⟩} ⟨вещественное⟩
⟨тег целого⟩ ::= **цел32** | **цел64**
⟨тег вещественного⟩ ::= **вещ32** | **вещ64** | **вещ128**
⟨пустой объект⟩ ::= **пусто32** | **пусто64** | **пусто** ⟨вычисление типа⟩
⟨строка⟩ ::=
стр1 ⟨поэлементное представление⟩ . . .
| **стр4** ⟨поэлементное представление⟩ . . .
| **стр8** ⟨поэлементное представление⟩ . . .

4. Генератор объекта.

⟨генератор⟩ ::=
⟨генератор объекта⟩ | ⟨генератор массива⟩
| ⟨генератор ситуации⟩ | ⟨генератор объекта связи⟩
| ⟨генератор внешнего объекта⟩ | ⟨генератор паспорта⟩
⟨генератор объекта⟩ ::=
ген ⟨вычисление типа⟩ {[⟨список атрибутов⟩]}
{**иниц** ⟨доопределение объекта⟩}
⟨доопределение объекта⟩ ::=
⟨доопределение составного⟩ | ⟨доопределение модуля⟩
| ⟨доопределение процедуры⟩
⟨атрибут⟩ ::= *локал*: ⟨выражение⟩ | *вгвект*: ⟨выражение⟩

4.1. Доопределение структур и массивов.

⟨доопределение составного⟩ ::=
⟨выражение⟩ | (⟨список доопределения элементов⟩)
⟨доопределение элемента⟩ ::=
{{⟨обозначение⟩:,} ... ⟨обозначение⟩} : ⟨выражение⟩
| {{⟨обозначение⟩:,} ... ⟨обозначение⟩} : ⟨доопределение объекта⟩
| [⟨диапазон⟩] : ⟨доопределение объекта⟩
⟨обозначение⟩ ::= ⟨идентификатор⟩ | ⟨выражение⟩

4.2. Генератор массива.

⟨генератор массива⟩ ::=
⟨генератор в-массива⟩ | ⟨генератор с-вектора⟩
⟨генератор в-массива⟩ ::=
⟨локализация⟩ [⟨список выражений⟩] ⟨формат⟩
⟨генератор с-вектора⟩ ::=
⟨локализация⟩ **вект** [⟨выражение⟩{ : ⟨выражение⟩}] ⟨формат⟩
⟨локализация⟩ ::= **лок** | **глоб**

5. Описания.

⟨описание⟩ ::=
| ⟨описание простых констант⟩ | ⟨описание текстов⟩
| ⟨описание простых переменных⟩ | ⟨описание типа⟩
| ⟨описание статических ситуаций⟩ | ⟨описание полей⟩
| ⟨описание меток⟩

5.1. Простые константы.

⟨описание простых констант⟩ ::= ⟨описание констант⟩
⟨описание констант⟩ ::=
конст ⟨список идентификации констант⟩
| **процедура** ⟨список идентификации процедур⟩
⟨идентификация константы⟩ ::=
⟨полная идентификация⟩ | ⟨предварительная идентификация⟩
| ⟨идентификация доопределения⟩
⟨полная идентификация⟩ ::=
⟨идентификатор⟩ {#⟨вычисление типа⟩} = ⟨выражение⟩
| ⟨идентификатор⟩ : ⟨вычисление типа⟩ =
⟨доопределение объекта⟩
⟨предварительная идентификация⟩ ::=
⟨идентификатор⟩
| {⟨идентификатор⟩#,} ...
⟨идентификатор⟩#⟨вычисление типа⟩
| {⟨идентификатор⟩:,} ...
⟨идентификатор⟩ : ⟨вычисление типа⟩

⟨идентификация доопределения⟩ ::=
⟨идентификатор⟩ = ⟨выражение⟩
| ⟨идентификатор⟩ = ⟨доопределение объекта⟩

5.2. Простые переменные.

⟨описание простых переменных⟩ ::= ⟨описание переменных⟩
⟨описание переменных⟩ ::=
перем ⟨список идентификации переменных⟩
| ⟨простой формат⟩ ⟨список идентификации переменных⟩
⟨идентификация переменной⟩ ::=
⟨полная идентификация переменной⟩
| ⟨предварительная идентификация переменной⟩
⟨предварительная идентификация переменной⟩ ::=
⟨идентификатор⟩
| {⟨идентификатор#,} . . .
⟨идентификатор⟩#⟨вычисление типа⟩
⟨полная идентификация переменной⟩ ::=
⟨идентификатор⟩ {#⟨вычисление типа⟩} := ⟨выражение⟩
| ⟨идентификатор⟩ = ⟨идентификатор⟩

5.3. Контекстная приставка.

⟨контекст⟩ ::= **контекст** {**огр**} ⟨секция контекста⟩
⟨секция контекста⟩ ::=
⟨последовательность описаний или включений⟩
⟨описание или включение⟩ ::=
⟨описание⟩ | ⟨включение в контекст⟩
⟨включение в контекст⟩ ::= ⟨имя модуля⟩
⟨имя модуля⟩ ::=
⟨идентификатор⟩ | ⟨элемент модуля⟩ | ⟨внешнее имя⟩

5.4. Доопределение модуля.

⟨доопределение модуля⟩ ::=
⟨тело модуля⟩
|(⟨выражение⟩ {, ⟨список выражений⟩})
⟨тело модуля⟩ ::=
⟨контекст⟩
начало ⟨последовательность описаний⟩
{; ⟨последовательность операторов⟩}
конец

5.5. Доопределение процедуры.

⟨доопределение процедуры⟩ ::=
{⟨контекст⟩} ⟨текст процедуры⟩
| (⟨выражение⟩ {, ⟨список выражений⟩})

6.1. Элемент массива.

⟨элемент массива⟩ ::= ⟨первичное⟩
[{⟨формат⟩} ⟨список выражений⟩]

6.2. Элемент структуры.

⟨элемент структуры⟩ ::= ⟨первичное⟩.⟨идентификатор⟩

6.3. Элемент модуля.

⟨элемент модуля⟩ ::=
⟨идентификатор⟩.⟨идентификатор⟩
| ⟨элемент модуля⟩.⟨идентификатор⟩

6.4. Подмассив.

⟨подмассив⟩ ::= ⟨первичное⟩ [{⟨формат⟩} ⟨диапазон⟩]
⟨диапазон⟩ ::= {⟨выражение⟩} : {⟨выражение⟩}

7. Выражения.

⟨выражение⟩ ::=
⟨формула⟩ | ⟨присваивание⟩ | ⟨операция над массивами⟩

7.1. Формула.

⟨формула⟩ ::=
⟨логическая сумма⟩ {**экв** ⟨логическая сумма⟩} . . .
⟨логическая сумма⟩ ::=
⟨логическое произведение⟩ {**или**
⟨логическое произведение⟩} . . .
⟨логическое произведение⟩ ::= ⟨отношение⟩{**и** ⟨отношение⟩} . . .
⟨отношение⟩ ::=
⟨сумма⟩ { ⟨операция отношения⟩ ⟨сумма⟩} . . .
| ⟨сравнение строк⟩
⟨сумма⟩ ::=
⟨произведение⟩ { ⟨операция группы сложения⟩
⟨произведение⟩} . . .
⟨произведение⟩ ::=
⟨сомножитель⟩ { ⟨операция группы умножения⟩
⟨сомножитель⟩} . . .
⟨сомножитель⟩ ::=
⟨одноместная формула⟩ { ⟨операция степени⟩
⟨одноместная формула⟩} . . .
⟨одноместная формула⟩ ::=
{⟨одноместная операция⟩} ⟨первичное⟩
| ⟨формирование указателя⟩
⟨первичное⟩ ::=
| ⟨изображение⟩ | ⟨идентификатор⟩
| ⟨элемент структуры⟩ | ⟨элемент модуля⟩
| ⟨элемент массива⟩ | ⟨подмассив⟩

| ⟨квалификация⟩ | ⟨вызов⟩
| ⟨генератор⟩ | ⟨закрытое выражение⟩
| ⟨формирование паспорта⟩ | ⟨формирование⟩
| ⟨форматный обмен⟩ | ⟨признак⟩
| ⟨подстановка текста⟩ | ⟨запрос атрибута⟩
| ⟨модификация атрибута⟩ | ⟨внешнее имя⟩
| ⟨двоичный обмен⟩ | ⟨поле значения⟩
| ⟨указуемая переменная⟩

⟨изображение⟩ ::=
⟨число⟩ | ⟨набор⟩
| ⟨статический вектор⟩ | ⟨текст процедуры⟩
| ⟨статический файл⟩ | ⟨статический справочник⟩
| ⟨пустой объект⟩

7.2. Приоритет операций.

⟨операция отношения⟩ ::=
⟨операция сравнения⟩ | <=> | **среди**
⟨операция сравнения⟩ ::=<> | = | > | >= | < | <=
⟨операция группы сложения⟩ ::= + | − | +: | −:
⟨операция группы умножения⟩ ::=* | **умнд** |*: | /: | **остат**
⟨операция степени⟩ ::= **
⟨одноместная операция⟩ ::=

целокр	**целобр**	**вещокр**	**вещобр**
ф32окр	**ф32обр**	**ф64окр**	**ф64обр**
цел64окр	**цел64обр**	**в128**	**целзн**
стнаб	**млнаб**	**естьнабор**	**естьпусто**
естьцел	**естьвещ**	**естьф32**	**естьф64**
естьф128	**не**	**пче**	**перв1**
знак	**длина**	**адрес**	**тип**
дес	**бит**	**тег**	

7.11. Квалификация.

⟨квалификация⟩ ::= ⟨первичное⟩#⟨вычисление типа⟩

8.1. Переменная.

⟨переменная⟩ ::=
⟨идентификатор⟩ | ⟨элемент массива⟩
| ⟨элемент структуры⟩ | ⟨элемент модуля⟩
| ⟨указуемая переменная⟩ | ⟨поле переменной⟩
| ⟨закрытая переменная⟩ | ⟨подстановка текста⟩

8.2. Присваивание.

⟨присваивание⟩ ::=
⟨переменная⟩ ::= ⟨выражение⟩

9. Закрытые предложения.

⟨закрытое предложение⟩ ::=
  {блок} {⟨контекст⟩}
  ⟨небазированное закрытое предложение⟩
⟨небазированный закрытый оператор⟩ ::=
    ⟨замкнутый оператор⟩ | ⟨условный оператор⟩
  | ⟨выбирающий оператор⟩ | ⟨цикл⟩
  | ⟨структурный оператор⟩ | ⟨выбирающее по типу⟩
⟨небазированное закрытое выражение⟩ ::=
    ⟨замкнутое выражение⟩ | ⟨условное выражение⟩
  | ⟨выбирающее выражение⟩ | ⟨структурное выражение⟩
⟨небазированная закрытая переменная⟩ ::=
    ⟨замкнутая переменная⟩ | ⟨условная переменная⟩
  | ⟨выбирающее переменную⟩
⟨последовательное предложение⟩ ::=
  {⟨описание⟩;} . . .{⟨помеченный оператор⟩;} . . .
  ⟨помеченное предложение⟩
⟨помеченное предложение⟩ ::= {⟨метка⟩} . . . ⟨предложение⟩
⟨оператор⟩ ::=
    ⟨присваивание⟩ | ⟨модификация атрибутов⟩
  | ⟨вызов⟩ | ⟨генератор внешнего объекта⟩
  | ⟨структурный переход⟩ | ⟨запуск задачи⟩
  | ⟨операции над массивами⟩ | ⟨подстановка текста⟩
  | ⟨закрытый оператор⟩ | ⟨двоичный обмен⟩
  | ⟨форматный обмен⟩ | ⟨формирование паспорта⟩
  | ⟨переход⟩

9.1. Замкнутое предложение.

⟨замкнутое предложение⟩ ::=
    (⟨последовательное предложение⟩)
  | начало ⟨последовательное предложение⟩ конец

9.2. Условное предложение.

⟨условное предложение⟩ ::=
    если ⟨условие⟩ то ⟨последовательное предложение⟩
  {иначе если ⟨условие⟩ то ⟨последовательное предложение⟩} . . .
  {иначе ⟨последовательное предложение⟩}
    все
⟨условие⟩ ::=
  {⟨описание⟩;} . . . {⟨помеченный оператор⟩;} . . .
  ⟨помеченное выражение⟩

9.3. Выбирающее предложение.

⟨выбирающее предложение⟩ ::=
    выбор ⟨вычисление номера⟩ из

⟨список размеченных альтернативных предложений⟩
иначе ⟨последовательное предложение⟩}
всевыб
| **выбор** ⟨вычисление номера⟩ **из**
⟨список последовательных предложений⟩
{**иначе** ⟨последовательное предложение⟩}
всевыб

⟨вычисление номера⟩ ::=
{⟨описание⟩;} . . . {⟨помеченный оператор⟩;} . . .
⟨помеченное выражение⟩

⟨размеченное альтернативное предложение⟩ ::=
{⟨номер⟩:,} . . . ⟨номер⟩: ⟨последовательное предложение⟩

⟨номер⟩ ::= ⟨выражение⟩

⟨выбирающее по типу⟩ ::=
выбтип ⟨выражение⟩ **из**
⟨список размеченных альтернатив типа⟩
{**иначе** ⟨последовательное предложение⟩}
всевыб

⟨размеченная альтернатива типа⟩ ::=
⟨описание альтернативы⟩ : ⟨последовательное предложение⟩

⟨описание альтернативы⟩ ::=
⟨идентификатор⟩#⟨вычисление типа⟩
| ⟨идентификатор⟩#(⟨параметр⟩)

⟨параметр⟩ ::= ⟨выражение⟩

9.4. Цикл.

⟨цикл⟩ ::=
{⟨заголовок цикла⟩}
цикл
{⟨описание⟩;} . . .
{⟨помеченный оператор⟩;} . . .
⟨помеченный оператор⟩
повторить

⟨заголовок цикла⟩ ::=
для ⟨идентификатор⟩ {**от** ⟨выражение⟩} ⟨компонента-до⟩
{**шаг** ⟨выражение⟩}
| **от** ⟨выражение⟩ ⟨компонента-до⟩

⟨компонента-до⟩ ::= **до** ⟨выражение⟩ | **вниздо** ⟨выражение⟩

10.1. Структурное предложение.

⟨структурное предложение⟩ ::=
до ⟨список определений ситуаций⟩
⟨закрытое предложение⟩
при ⟨список альтернативных предложений⟩ **всесит**

⟨описание статических ситуаций⟩ ::=
статсит ⟨список идентификаторов⟩

⟨генератор ситуации⟩ ::=
ситуация (⟨список установки атрибутов⟩)
⟨альтернативное предложение⟩ ::=
{⟨идентификатор⟩:,} . . . ⟨идентификатор⟩
{(⟨список предварительных идентификаций⟩)}
: ⟨последовательное предложение⟩

10.2. Структурный переход.

⟨структурный переход⟩ ::= ⟨первичное⟩ ! { (⟨список выражений⟩)}

11.1. Процедура.

⟨текст процедуры⟩ ::=
⟨спецификация процедуры⟩ ⟨закрытый оператор⟩
| ⟨спецификация процедуры⟩ ⟨закрытое выражение⟩
| ⟨реализация процедуры⟩ | ⟨реализация модуля⟩
⟨спецификация процедуры⟩ ::=
проц (⟨параметры⟩)
| **функция** (⟨параметры⟩)
| ⟨тип процедуры⟩
⟨параметры⟩ ::= ⟨последовательность описаний формальных⟩
⟨описание формального⟩ ::=
{⟨простой формат⟩} ⟨список идентификаторов⟩
| ⟨описание базы⟩
⟨идентификация процедуры⟩ ::=
⟨идентификатор⟩ { = ⟨доопределение процедуры⟩}
⟨вызов⟩ ::= ⟨первичное⟩ ({ ⟨список фактических⟩})
⟨фактический⟩ ::=
⟨выражение⟩ | ⟨доопределение объекта⟩
| **прог** ⟨выражение⟩ | **имя** ⟨переменная⟩
| **имя прог** ⟨переменная⟩
⟨реализация модуля⟩ ::=
{⟨формальный контекст⟩}
реал ⟨имя типа⟩
{(⟨список идентификации констант⟩)} ⟨тело модуля⟩
⟨формальный контекст⟩ ::=
контекст (⟨список идентификации констант⟩)
⟨реализация процедуры⟩ ::=
{⟨формальный контекст⟩}
реал {⟨контекст⟩} ⟨текст процедуры⟩

11.3. Задача.

⟨запуск задачи⟩ ::=
задача (⟨список установок атрибутов⟩)
⟨текст программы⟩ { (⟨список фактических⟩)}

12.1. Программа-объект.

⟨текст программы⟩ ::=
⟨описание константы⟩ | ⟨описание типа⟩
| ⟨закрытый оператор⟩ | ⟨программа-вычисление⟩
⟨программа-вычисление⟩ ::=
программа {⟨описание⟩ ;} ... {⟨оператор⟩;} ... ⟨выражение⟩
конец

13.1. Операции форматного обмена.

⟨форматный обмен⟩ ::=
запф (⟨выражение⟩, ⟨список элементов вывода⟩)
| *читф* (⟨выражение⟩, ⟨список элементов ввода⟩)
| *запфм* (⟨массив обмена⟩, ⟨список элементов вывода⟩)
| *читфм* (⟨массив обмена⟩, ⟨список элементов ввода⟩)
⟨элемент вывода⟩ ::=
⟨выражение⟩ {: ⟨формат обмена⟩} | : ⟨позиционирование⟩
⟨элемент ввода⟩ ::=
имя ⟨переменная⟩ { : ⟨формат обмена⟩}
| ⟨выражение⟩ { : ⟨формат обмена⟩}
| : ⟨позиционирование⟩
⟨массив обмена⟩ ::= ⟨выражение⟩ | **имя** ⟨переменная⟩

13.3. Позиционирование.

⟨позиционирование⟩ ::= ⟨литерал⟩ { ⟨размещение⟩} ...
|⟨размещение⟩ ...
⟨размещение⟩ ::= {⟨повторитель⟩}⟨код размещения⟩ { ⟨литерал⟩}
⟨литерал⟩ ::=
{⟨повторитель⟩} "⟨литера⟩..."
{⟨повторитель⟩"⟨литера⟩..."}...
⟨код размещения⟩ ::= *x* | *y* | *к* | *l* | *p*
⟨повторитель⟩ ::= ⟨целое⟩ | *n* (⟨выражение⟩)
здесь *n* — латинское *N*.

13.4. Обмен, управляемый форматом.

⟨формат обмена⟩ ::= {⟨вставка⟩} ⟨трафарет⟩
⟨трафарет⟩ ::= ⟨трафарет числа⟩ | ⟨трафарет целого⟩
| ⟨трафарет вещественного⟩ | ⟨трафарет литерного⟩
| ⟨трафарет логического⟩ | ⟨трафарет двоичного⟩
| ⟨трафарет тегированного⟩
⟨вставка⟩ ::= ⟨литера⟩{⟨смещение⟩} ... | ⟨смещение⟩...
⟨смещение⟩ ::= {⟨повторитель⟩} *x* {⟨литерал⟩}
⟨трафарет числа⟩ ::=
{⟨повторитель⟩} *g* {(⟨выражение⟩ {, ⟨выражение⟩
{, ⟨выражение⟩})} {⟨вставка⟩}

⟨трафарет целого⟩ ::= {⟨поле знака⟩} поле ⟨цифр⟩
⟨поле знака⟩ ::=
⟨знак⟩ {⟨вставка⟩} | ⟨поле плавающего знака⟩ {⟨вставка⟩}
⟨поле цифр⟩ ::= ⟨подполе цифр⟩...
⟨подполе цифр⟩ ::=
{⟨повторитель⟩} *s* *d* {⟨вставка⟩}
| {⟨повторитель⟩} *z* {⟨условная вставка⟩}
| {⟨повторитель⟩} *z* ⟨вставка⟩
⟨поле плавающего знака⟩ ::=
⟨подполе плавающего знака⟩... ⟨знак⟩
⟨подполе плавающего знака⟩ ::=
{⟨повторитель⟩} *f* {⟨условная вставка⟩}
| {⟨повторитель⟩} *f* ⟨вставка⟩
⟨условная вставка⟩ ::= *o* ⟨вставка⟩
⟨трафарет вещественного⟩ ::=
⟨трафарет с фиксированной точкой⟩
| ⟨трафарет с плавающей точкой⟩
⟨трафарет с фиксированной точкой⟩ ::=
⟨трафарет целого⟩ ⟨поле точки⟩ ⟨поле цифр⟩
⟨трафарет с плавающей точкой⟩ ::=
⟨трафарет с фиксированной точкой⟩ ⟨поле порядка⟩
| ⟨трафарет целого⟩ ⟨поле порядка⟩
⟨поле точки⟩ ::= {*s*}.{⟨вставка⟩}
⟨поле порядка⟩ ::= {*s*} *e* ⟨трафарет целого⟩
⟨трафарет логического⟩ ::= *b* {⟨вставка⟩}
⟨трафарет литерного⟩ ::= ⟨подполе литер⟩...
⟨подполе литер⟩ ::= {⟨повторитель⟩} {*s*} *a* {⟨вставка⟩}
⟨трафарет двоичного⟩ ::= ⟨основание⟩ *r* ⟨поле цифр⟩
{⟨вставка⟩}
⟨основание⟩ ::= 1 | 3 | 4
⟨трафарет тегированного⟩ ::= ⟨основание⟩ *t* {⟨вставка⟩}

14.1. Пересылка.

⟨пересылка⟩ ::=
⟨вектор назначения⟩ ⟨операция пересылки⟩
⟨источник пересылки⟩ {&& ⟨источник пересылки⟩}
⟨источник пересылки⟩ ::=
⟨безусловная пересылка⟩ | ⟨условная пересылка⟩
| ⟨перевод⟩ | ⟨распаковка⟩ | ⟨заполнение⟩
⟨вектор назначения⟩ ::= ⟨формула⟩ | **мод** ⟨переменная⟩
⟨операция пересылки⟩ ::= <:=
⟨безусловная пересылка⟩ ::=
{⟨простой формат⟩} ⟨вектор источника⟩
{**длиной** ⟨максколичество⟩}
⟨условная пересылка⟩ ::=

⟨безусловная пересылка⟩ **пока** ⟨индикатор отношения⟩
⟨первичное⟩
⟨вектор источника⟩ ::= ⟨первичное⟩ | **мод** ⟨переменная⟩
⟨максколичество⟩ ::= ⟨первичное⟩ | **мод** ⟨переменная⟩
⟨индикатор отношения⟩ ::= ⟨операция сравнения⟩
| **среди** | **не среди**
⟨заполнение⟩ ::=
{⟨простой формат⟩} **заполн** ⟨первичное⟩
{**длиной** ⟨максколичество⟩}
⟨перевод⟩ ::=
перевод ⟨вектор источника⟩ {**длиной** ⟨максколичество⟩}
по ⟨первичное⟩
⟨распаковка⟩ ::=
⟨операция распаковки⟩ ⟨первичное⟩ {**длиной** ⟨максколичество⟩}
⟨операция распаковки⟩ ::= **распак** | **распакзн**

14.2. Операции поиска.

⟨поиск⟩ ::= ⟨поиск по строке⟩ | ⟨поиск по маске⟩
⟨поиск по строке⟩ ::= ⟨указатель источника⟩
от ⟨условие поиска⟩
⟨условие поиска⟩ ::=
⟨индикатор отношения⟩ ⟨первичное⟩
| ⟨максколичество⟩ {**либо** ⟨индикатор отношения⟩
⟨первичное⟩}
⟨указатель источника⟩ ::= ⟨формула⟩ | **мод** ⟨переменная⟩
⟨поиск по маске⟩ ::=
поиск (⟨выражение⟩, ⟨выражение⟩
{,{⟨выражение⟩,} ⟨выражение⟩})
⟨способ выборки⟩ ⟨операция сравнения⟩ ⟨первичное⟩
⟨способ выборки⟩ ::= **до** | **вниздо** | **спискомдо** | **цепьюдо**

14.3. Сравнение строк.

⟨сравнение строк⟩ ::=
⟨вектор источника⟩/⟨операция сравнения⟩
/ ⟨вектор источника⟩
{**длиной** ⟨максколичество⟩}
⟨операция сравнения⟩ ::= < | > | = | <> | <= | >=

14.5. Упаковка.

⟨упаковка⟩ ::= **пак** ⟨вектор источника⟩
{**длиной** ⟨максколичество⟩}

14.6. Признаки.

⟨признак⟩ ::= **тго** | **тгп** | **тги**

15.1. Формирование паспорта.

⟨формирование паспорта⟩ ::=
  формавм ({⟨описатель⟩} [⟨список идентификаторов⟩] =
    ⟨описатель⟩ [⟨список диапазонов или индексов⟩])
⟨описатель⟩ ::= ⟨идентификатор⟩ | ⟨закрытое выражение⟩
⟨диапазон или индекс⟩ ::= ⟨суперпозиция⟩ {= ⟨диапазон⟩}
⟨суперпозиция⟩ ::= ⟨слагаемое⟩ {⟨знак⟩ ⟨слагаемое⟩} . . .
⟨слагаемое⟩ ::= ⟨сомножитель⟩ {* ⟨сомножитель⟩}

15.3. Генератор паспорта.

⟨генератор паспорта⟩ ::=
    ⟨локализация⟩ [{, } . . .]
  | ⟨локализация⟩ [⟨список выражений⟩]

16. Участок базированной области.

⟨описание базы⟩ ::= **база** ⟨идентификатор⟩

17. Текстовые макросы.

⟨описание текстов⟩ ::= **текст** ⟨список идентификации текстов⟩
⟨идентификация текста⟩ ::=
  ⟨идентификатор⟩ {({⟨параметры текста⟩})}
  {# ⟨вычисление типа⟩} = ⟨предложение⟩
⟨подстановка текста⟩ ::=
    ⟨идентификатор⟩ {({⟨список предложений⟩})}
  | ⟨идентификатор⟩ [⟨список предложений⟩]
⟨параметры текста⟩ ::= ⟨список предварительных идентификаций⟩

18. Атрибуты объектов.

⟨запрос атрибута⟩ ::=
  читатр (⟨выражение⟩, {⟨уточнение⟩,} ⟨выражение⟩)
⟨уточнение⟩ ::= ⟨выражение⟩
⟨модификация атрибутов⟩ ::=
  вапатр (⟨выражение⟩, {⟨уточнение⟩,}
  ⟨список установок атрибутов⟩)
⟨установка атрибута⟩ ::= ⟨выражение⟩ : ⟨выражение⟩

19. Описание и использование меток.

⟨описание меток⟩ ::= **метка** ⟨список идентификаторов⟩
⟨метка⟩ ::= ⟨идентификатор⟩ ^:
⟨переход⟩ ::= **на** ⟨одноместная формула⟩

Глава 3.

2.1. Генератор объекта.

⟨генератор внешнего объекта⟩ ::=
  ⟨спецификация внешнего⟩ ({⟨выражение⟩,} ⟨выражение⟩
  {, ⟨список установок атрибутов⟩})

⟨спецификация внешнего⟩ ::=
генфайл | *генконтейнер* | *генспр*
⟨генератор объекта связи⟩ ::=
⟨спецификация объекта связи⟩
({⟨выражение⟩,} ⟨список установок атрибутов⟩)
| ⟨спецификация объекта связи⟩
({⟨список установок атрибутов⟩})
⟨спецификация объекта связи⟩ ::=
файл | *позп* | *тбуф* | *бвв* | *контейнер* | *прогр* | *генув*

5.4. Внешнее имя.

⟨внешнее имя⟩ ::= {⟨идентификатор⟩} ⟨план поиска⟩
⟨план поиска⟩ ::= ⟨многослоговое имя⟩ | // [⟨выражение⟩]
⟨многослоговое имя⟩ ::= ⟨слог⟩ . . .
⟨слог⟩ ::= // ⟨буква или цифра⟩. . . | // ⟨стандартный слог⟩
⟨стандартный слог⟩ ::= *.обкт* | *.текст* | *.код* | *.прогр*

7. Обмен. Общие сведения

⟨двоичный обмен⟩ ::=
⟨идентификатор операции⟩ (⟨выражение⟩,
⟨спецификация адреса⟩ {, ⟨выражение⟩}
{, ⟨*ошобмена*: ⟨выражение⟩})
⟨идентификатор операции⟩ ::=
чит | *читпар* | *зап* | *заппар* | *нов* | *уст* | *устпар*
⟨спецификация адреса⟩ ::=
⟨выражение⟩
| ⟨направление⟩ {(⟨выражение⟩)}
| ⟨выражение⟩, ⟨направление⟩ {(⟨выражение⟩)}
| *нач* | *кон* | *канал* (⟨выражение⟩)
⟨направление⟩ ::= *след* | *пред*

13. Изображение файла

⟨статический файл⟩ ::=
⟨изображение произвольного файла⟩
| ⟨изображение текстового файла⟩
⟨изображение произвольного файла⟩ ::=
файл ⟨вложенный файл⟩
⟨изображение текстового файла⟩ ::=
тфайл [(⟨список установок атрибутов⟩)]
⟨определение ограничителя⟩ ⟨текст файла⟩
⟨ограничитель⟩

14. Статический справочник

⟨статический справочник⟩ ::=
спрконст (⟨список элементов справочника⟩)
⟨элемент справочника⟩ ::= ⟨идентификатор⟩ {@} = ⟨выражение⟩

Глава 5.

2. Преобразование типа массива.

⟨формирование указателя⟩ ::= @ ⟨переменная⟩
⟨указуемая переменная⟩ ::= ⟨первичное⟩ @
⟨массив констант⟩ ::=
  мконст ⟨формат⟩
  (⟨список элементов массива констант⟩)
⟨элемент массива констант⟩ ::=
    {⟨число повторений⟩} ⟨выражение⟩
  | {⟨число повторений⟩}
  (⟨список элементов массива констант⟩)
⟨число повторений⟩ ::= / ⟨целое⟩ /

4. Описание и использование полей.

⟨поле переменной⟩ ::= ⟨переменная⟩.⟨описатель поля⟩
⟨описатель поля⟩ ::=
    ⟨непосредственный описатель поля⟩
  | ⟨идентификатор⟩
⟨непосредственный описатель поля⟩ ::=
  [{⟨строчный формат⟩} ⟨выражение⟩
  {: {⟨выражение⟩}}]
⟨формирование⟩ ::=
  ⟨первичное⟩.(⟨список элементов формирования⟩)
⟨элемент формирования⟩ ::=
  ⟨описатель поля⟩ : ⟨выражение⟩ | **тип** : ⟨выражение⟩

5.4. Поиск с маскированием

⟨поиск по маске⟩ ::=
  поиск (⟨выражение⟩, ⟨выражение⟩
  {, {⟨выражение⟩,} ⟨выражение⟩})
  ⟨способ выборки⟩ ⟨операция сравнения⟩ ⟨первичное⟩
⟨способ выборки⟩ ::=
  до | **вниздо** | **спискомдо** | **цепьюдо**

7. Размещение элементов структуры.

⟨спецификация размещения⟩ ::=
  [⟨формат⟩ {⟨выражение⟩} : {⟨выражение⟩}]

ПРИЛОЖЕНИЕ 2

ИНТЕРФЕЙС МОДУЛЯ СТАНДАРТНОГО КОНТЕКСТА ПОЛЬЗОВАТЕЛЯ

В комментарии к описанию идентификатора указан номер страницы, где изложена семантика данного идентификатора стандартного контекста.

```
интерфейс
тип
    % 1. Логический тип и типы наборов.
лог; % скрытый тип
тип
наб = выбтип до 64 из %*119%*
    0 : массив [0] конст#лог,
    1 : лог,
    2 : массив[2]конст#лог,
    % и т. д., общий вид альтернатив:
    % к: массив[к]конст#лог,
    64 : массив[64]конст#лог
      всевыб,
циф = наб (4), %*119%*
лит = наб (8), %*119%*
    % 2. Числовые типы %120, 121%
    % 2.1. Целые числа.
цел = структ(конст вн#лог, мант : наб(31)),
    % целое формата ф64:
дцел = структ(конст зн#лог, мант : наб(63)),
    % универсальный класс целых формата ф32 и ф64:
тунц = (тунцц, тунцдц, тунцн),
унцел =
выбтип тунц из тунцц : цел, тунцдц : дцел, тунцн : наб всевыб;
    % максимальные целые числа:
конст максцел#цел, максдцел#дцел,
    % 2.2. Вещественные числа.
    % длины мантисс вещественных чисел:
длмквещ#цел=24, длмвещ# цел=56;
    % короткое вещественное формата ф32:
тип
квещ = структ(констзн#лог, порядок : цел,
                мант : наб (длмквещ))
    % вещественное формата ф64:
вещ = структ(конст вн#лог, порядок : цел,
                мант : наб(длмвещ)),
    % длинное вещественное формата ф128:
двещ = структ(конст вн#лог, порядок : цел,
```

```
    мантмл :, мантст : наб(длмвещ));
    % максимальные вещественные числа и
    % вещественные нули:
конст максквещ#, минквещ#квещ,
      максвещ#, минвещ#вещ,
      максдвещ#, миндвещ#двещ;
    % 2.3. Числовые типы.
    % короткое числовое формата ф32:
тип
ткч = (ткчц, ткчкв),
кчис=выбтип ткч из ткчц : цел, ткчкв : квещ всевыб,
    % числовое формата ф64:
тч = (тчдц, тчв),
чис = выбтип тч из тчдц : дцел, тчв : вещ всевыб,
    % универсальный числовой тип всех форматов:
тунч =(тунчкч, тунчч, тунчн, тунчдв),
унчис = выбтип тунч из
          тунчкч : кчио, тунчч : чис,
          тунчн : наб, тунчдв : двещ
        всевыб,
    % 3. Стандартные типы массивов.
тэм = (тлог, тциф, тлит, ткор, тдин, тдлин),
    % 3.1. Тип стандартного вектора.
стстр = выбтип тэм из %*120%*
    % типы с-векторов констант
    тлог : массив[ ]конст#лог,
    тциф : массив[ ]конст#циф,
    тлит : массив[ ]конст#лит,
    ткор : массив[ ]конст#кчис,
    тдин : массив[ ]конст,
    тдлин : массив[ ]конст#унчис
    всевыб,
стсв = выбтип тэм из %*120%*
    % типы с-векторов переменных
    тлог : массив[ ]ф1,
    тциф : массив[ ]ф4,
    тлит : массив[ ]ф8,
    ткор : массив[ ]ф32,
    тдин : массив[ ]ф64,
    тдлин : массив[ ]ф128
    всевыб
тствект = (тстр, тсв),
ствект = выбтип тствект из %*120%*
    тстр : стстр,
```

```
    тсв   : стсв
     всевыб,
    % 3.2. Типы выстроенных массивов:
вм1 = выбтип тэм из %*121%*
    тлог  : массив[    ]ф1,
    тциф  : массив[    ]ф4,
    тлит  : массив[    ]ф8,
    ткор  : массив[    ]ф32,
    тдин  : массив[    ]ф64,
    тдлин : массив[    ]ф128
     всевыб,
    % аналогично строятся описания типов
    % двумерного массива вм2 и т. д. до вм7.
ствм = выбтип до 7 из %*121%*
    1 : вм1, 2 : вм2, 3 : вм3, 4 : вм4, 5 : вм5, 6 : вм6, 7 : вм7
     всевыб,
    % 3.3. Сокращения для программ,
    % использующих массивы стандартных типов
стрлог = стстр (тлог), стрциф = стстр (тциф),
стрлит = стстр (тлит), стркор = стстр (ткор),
стрдин = стстр (тдин), стрдлин = стстр (тдлин),
свлог = стсв (тлог), свциф = стсв (тциф),
свлит = стсв (тлит), свкор = стсв (ткор),
свдин = стсв (тдин), свдлин = стсв (тдлин),
вм2лог = вм2 (тлог), вм2циф = вм2 (тциф),
вм2лит = вм2 (тлит), вм2кор = вм2 (ткор),
вм2дин = вм2 (тдин), вм2длин = вм2 (тдлин),
    % 4. Стандартные скрытые типы:
стпроц, % стандартный тип процедуры, 124, 166
ствж, % объект, определяющий локализацию, 126
стсит, % динамическая ситуация, 124, 160
стсем; % семафор, 125, 174
    % 5. Стандартные ситуации
конст
границамассива#, %*137, 335%*
делениенануль#, %*144%*
исчерпвремяопрц#, %*335%*
нарушениезащиты#, %*123%*
неверныйоперанд#, %*140, 148%*
ошибкаимяни#, %*170, 326%*
переполнвещ#, %*144%*
переполнвещ128#, %*144%*
переполнцел32#, %*144%*
переполнцел64#, %*144%*
привдействия#, %*105%*
ситнетархслк#, %*301%*
ситнетварх#, %*301%*
ситнетвнеш#, %*301%*
ситнетлиста#, %*300%*
ситнетмас#, %*127%*
ситнетрес#, %*301%*
ситошатр#, %*301%*
ситошву#, %*300%*
ситошпарамарх#, %*301%*
ситошпарамобмена#, %*301%*
ситошсмв#, %*300%*
ситпривобмен#, %*301%*
```

ситлкф##, %*300%*
ситфкф##стсит; %*300%*

% 6. Стандартные процедуры

процедура
генвж, %*126%*
ждать, %*176%*
вакрытьсем, %*176%*
вапвк, %*259%*
запспр, %*259%*
измдлину, %*127%*
измтип, %*144%*
копирув, %*256%*
конспр, %*260%*
ликвидбуф, %*250%*
ликвидэлспр, %*250%*
ликвидвнеш, %*250%*
начспр, %*260%*
обкт, %*255%*
откреп, %*251%*
открепбуф, %*290%*
открытьсем, %*176%*
перестлист, %*271%*
предэлспр, %*260%*
пввект, %*210%*
провтип, %*145%*
пропустить, %*177%*
сброс, %*291%*
сброспар, %*291%*
создпроцесс, %*176%*
следэлспр, %*260%*
создэлвк, %*251%*
создэлспр, %*259%*
тгз, %*206%*
текэлспр, %*260%*
элвк, %*258%*
элспр; %*258%*

% 7. Стандартные константы

конст
ацпу, %*249%*
барабан, %*249%*
глобарх, %*254%*
впк, %*249%*
впл, %*249%*
ксн, %*254%*
мдконт, %*249%*
млконт, %*249%*
нпо, %*254%*
пустовнеш, %*256%*
стконт; %*249%*

% Атрибуты

конст
адресзаписи, %*321%*
активность, %*313%*
базалиста, %*316%*
базвект, %*322%*
блоклкф, %*307%*
блокфайла, %*308, 318%*
виртвыв, %*323%*
внешконт, %*314%*
вобмене, %*320%*
восстановимый, %*312%*
времяобмена, %*314, 322%*
времяцп, %*322%*
датасозд, %*309%*
двоичный, %*306%*
длинблока, %*308, 318%*
длинизм, %*321%*
длинлиста, %*306%*
длинсвс, %*314%*
длинсектора, %*313%*
длинстрок, %*310%*
длинфнс, %*310%*
запасбуферов, %*313%*
запконтроль, %*315%*
имязадачи, %*323%*
имяконт, %*317%*
имяпол, %*315, 318, 323%*
имяфайла, %*308%*
имяэлспр, %*320%*
имяяз, %*311%*
инфпрогр, %*314%*
катлг, %*258%*
класслиста, %*315%*
классфайла, %*312%*
кодблока, %*306%*
кодпрогр, %*311%*
кодфнс, %*310%*
косвсак, %*320%*
листвцилиндр, %*309%*
локал, %*305%*

СПИСОК ЛИТЕРАТУРЫ

1. Пентковский В. М. Автокод Эльбрус (Эль-76). Принципы построения языка и руководство к пользованию.— М.: Наука, 1982.
2. Бабаян Б. А., Сахин Ю. Х. Система Эльбрус // Программирование.— 1980. № 6.— С. 72—86.
3. Общие принципы разработки архитектуры и программного обеспечения системы Эльбрус/Бабаян Б. А., Иванов А. П., Пентковский В. М., Плоткин А. Л., Семенихин С. В., Торчигин В. П., Чинин Г. Д.— Препринт/ИТМиВТ АН СССР.— М., 1985.— № 17.— 30 с.
4. Особенности архитектуры центрального процессора МВК Эльбрус/Бабаян Б. А., Горштейн В. Я., Ким Г. С., Назаров Л. Н., Сахин Ю. Х.— Препринт/ИТМиВТ АН СССР.— М., 1985.— № 15.— 46 с.
5. Система ввода—вывода МВК Эльбрус/Пшеничников Л. Е., Горшков П. В., Захватов М. В., Кольцова С. Л.— Препринт/ИТМ и ВТ АН СССР.— М., 1985.— № 19.— 16 с.
6. Основные принципы взаимодействия процессора передачи данных с системой МВК Эльбрус/Рябов Г. Г., Бабаян Б. А., Семенихин С. В., Перекатов В. Н., Ларин Е. М. // Автоматика и вычислительная техника.— 1987.— № 3. — С. 6—13.
7. Операционная система МВК Эльбрус/Зотов С. М., Иванов А. П., Семенихин С. В., Шевяков В. С.— Препринт/ИТМ и ВТ АН СССР.— М., 1985.— № 20.— 43 с.
8. Унифицированные механизмы управления физической памятью в операционной системе МВК Эльбрус/Еремин М. В., Иванов А. П., Шевяков В. С.— Препринт/ИТМ и ВТ АН СССР.— М., 1984.— № 9.— 48 с.
9. Ананьев Л. И. Перемещение задач в МВК Эльбрус.— Препринт/ИТМ и ВТ АН СССР.— М., 1985.— № 5.— 37 с.
10. Зотов С. М., Семенихин С. В. Взаимодействие процессов в МВК Эльбрус.— Препринт/ИТМ и ВТ АН СССР.— М., 1981.— № 1.— 21 с.
11. Зотов С. М., Семенихин С. В. Параллельные вычисления и механизмы синхронизации в МВК Эльбрус // Системное и теоретическое программирование: Тез. докл. 4 Всесоюзн. симпоз.— Кишинев: Кишиневский государственный университет, 1983.— С. 169—174.
12. Система простых файлов МВК Эльбрус/Торчигин В. П., Веретенников С. В., Ревякин В. А., Субботин А. А., Харитонов М. И.— Препринт/ИТМ и ВТ АН СССР.— М., 1985.— № 18.— 23 с.

13. Сумской В. В., Ананьев Л. И. Буферизованный обмен с коллективными файлами в МВК Эльбрус.— Препринт / ИТМ и ВТ АН СССР.— М. 1983.— № 9.— 25 с.
14. Особенности реализации системы простых файлов МВК Эльбрус/Торчинин В. П., Харитонов М. И., Авдеев С. Д., Веретенников С. В., Волков А. Н., Лебедев В. П., Пилипенко Е. О. // Системное и теоретическое программирование: Тез. докл. 4 Всесоюзн. симпоз.— Кишинев: Кишиневский государственный университет, 1983.— С. 371.
15. Основные принципы управления базами данных в МВК Эльбрус/Бабаян Б. А., Веретенников С. В., Лебедев В. П.— Препринт/ИТМ и ВТ АН СССР.— М., 1985.— № 17.— 17 с.
16. Харитонов М. И. Командный язык администратора вычислительного центра МВК Эльбрус.— Препринт / ИТМ и ВТ АН СССР.— М., 1980.— № 23.— 25 с.
17. Давыдов В. А., Мамай А. К., Мамай И. И. Основные особенности командного языка МВК Эльбрус // Конференция молодых ученых и специалистов ИТМ и ВТ: Тез. докл.— М.: ИТМ и ВТ АН СССР.— 1981. — С. 92—94.
18. Бабаян Б. А., Семенихин С. В. Организация сетей в МВК Эльбрус.— Препринт/ИТМ и ВТ АН СССР.— М., 1985. — № 21.— 16 с.
19. Бабаян Б. А., Волконский В. Ю., Пентковский В. М. Системная поддержка модульного программирования.— Препринт/ИТМ и ВТ АН СССР.— М., 1985.— № 11.— 72 с.
20. Колесник И. П., Малышев Н. Б., Пентковский В. М. Реализация модульных объектов в системе Эльбрус // Системное и теоретическое программирование: Тез докл. 4 Всесоюзн. симпоз.— Кишинев: Кишиневский государственный университет, 1983.— С. 207—208.
21. Механизмы реализации пакетов обработки абстрактных типов данных в МВК Эльбрус/Пентковский В. М., Колесник И. П., Малышев Н. Б., Крушняков В. Н. // Автоматизация производства систем программирования: Тез. докл. 3 Всесоюзн. конф.— Таллин: ИК АН ЭССР, 1986.— С. 172—173.
22. Пентковский В. М. Средства структурированного программирования для языка высокого уровня.— Препринт/ИТМ и ВТ АН СССР.— М., 1976.— № 6.— 25 с.
23. Транслятор автокода «Л»/Пентковский В. М., Белостоцкий А. Н., Волконский В. Ю., Лесовая И. Ф., Румянцев Ю. С., Чернис Е. Н.— Препринт/ИТМ и ВТ АН СССР.— М., 1975.— № 10.— 20 с.
24. Пентковский В. М., Гиндеев Б. П. Использование расширяемого языка высокого уровня для определения специализированных языков управления системами // Прикладная информатика.— 1984. Вып. 2.— С. 93—102.
25. Пентковский В. М. Языковые основы проектирования системы Эльбрус // Системное и теоретическое программирование: Тез. докл. 4 Всесоюзн. симпоз.— Кишинев.: Кишиневский государственный университет, 1983.— С. 294—296.
26. Бабаян Б. А., Пентковский В. М. Языковая модель системной поддержки модульного программирования.— Препринт/ИТМ и ВТ АН СССР.— М., 1985.— № 7.— 59 с.
27. Пентковский В. М. Влияние архитектуры системы Эльбрус на разработку систем программирования // Всесоюзная

конференция по методам трансляции: Тез. докл.— Новосибирск.: ВЦ СО АН СССР, 1981.— С. 59—60.

28. Основные компоненты системы программирования МВК Эльбрус и их взаимодействие/Ахрамеев А. В., Волконский В. Ю., Давидчук М. В., Добров А. Д., Зотов С. М., Малышев Н. Б., Пентковский В. М., Петровский С. Ю., Преображенский О. М., Приказчикова М. С., Румянцев Ю. С., Синдеев Б. П., Тухватулин Г. М., Цветкова Г. А., Чижова В. С.// Всесоюзная конференция по методам трансляции: Тез. докл.— Новосибирск: ВЦ СО АН СССР, 1981.— С. 56—58.

29. Волконский В. Ю., Пентковский В. М. Универсальный интерфейс компонентов системы программирования МВК Эльбрус.— Препринт/ИТМ и ВТ АН СССР.— М., 1980.— № 28.— 39 с.

30. Структурные модели программы и памяти для систем программирования/Волконский В. Ю., Еремин М. В., Пентковский В. М., Петровский С. Ю. // Автоматизация производства пакетов прикладных программ: Тез. докл. 2 Всесоюзн. конф.— Таллин: ИК АН ЭССР, 1983.— С. 112—114.

31. Опыт реализации и применения технологии создания систем программирования на базе языкоориентированной системы / Волконский В. Ю., Добров А. Д., Еремин М. В., Крушняков В. Н., Пентковский В. М., Румянцев Ю. С., Сушенцов А. Л., Чернис Е. Н. // Автоматизация производства систем программирования: Тез. докл. 3 Всесоюзн. конф.— Таллин: ИК АН ЭССР, 1986.— С. 154—156.

32. Волконский В. Ю., Крушняков В. Н., Сушенцов А. Л. Обеспечение мобильности трансляторов в МВК Эльбрус // Проблемы создания супер-ЭВМ, супер-систем и эффективность их применения: Тез. докл. 1 Всесоюзн. конф.— Минск, 1987.— С. 124—126.

33. Особенности организации доступа к файлам текста и кода в МВК Эльбрус/Волконский В. Ю., Колесник И. П., Мартынова Н. В., Цветкова Г. А. // Конференция молодых ученых и специалистов ИТМ и ВТ: Тез. докл.— М.: ИТМ и ВТ АН СССР, 1981.— С. 103—105.

34. Особенности комплексации программ в системе программирования МВК Эльбрус/Добров А. Д., Пентковский В. М., Приказчикова М. С., Чижова В.С. // Управляющие системы и машины.— 1982.— № 4.— С. 92—96.

35. Румянцев Ю. С., Синдеев Б. П. Проблемы реализации базового языка МВК Эльбрус // Высокопроизводительные вычислительные системы: Тез. докл. Всесоюзн. совещ.— М.: ИПУ АН СССР, 1981.— С. 93—94.

36. Румянцев Ю. С., Чернис Е. Н. Особенности оптимизаций в МВК Эльбрус // Методы трансляции и конструирования программ.— Новосибирск: ВЦ СО АН СССР, 1986.— С. 70—76.

37. Реализация Алгола-60 и Фортрана на МВК Эльбрус/Дятлова Л. Г., Кожухина О. К., Лесовая И. Ф., Пенькова Л. Н., Петрашкова Н. Ю., Сидорова Н. А., Хмельницкая О. В., Шишова Н. А.— Препринт/ ИТМ и ВТ АН СССР.— М., 1978.— № 21.— 16 с.

38. Окольнишников В. В., Шелехов В. И. Архитектура системы Алгол-Эльбрус // Многопроцессорные вы-

числительные системы и их математическое обеспечение.— Новосибирск: ВЦ СО АН СССР, 1982.— С. 59—66.
39. Мезенцев В. А., Петрашкова Н. Ю., Черкашина Н. А. Производительность Фортрана на МВК Эльбрус // Методы трансляции и конструирования программ.— Новосибирск: ВЦ СО АН СССР, 1986.— С. 66—69.
40. Андреев В. М., Голосов И. С., Самойленко С. М. Особенности проведения оптимизирующих преобразований // Методы трансляции и конструирования программ.— Новосибирск: ВЦ СО АН СССР, 1986.— С. 55—59.
41. Шелестов С. М. Реализация языка ПЛ/1 для МВК Эльбрус // Научно-технический семинар по программному обеспечению МВК Эльбрус: Тез. докл.— Новосибирск: НФ ИТМ и ВТ, 1981.— С. 13—14.
42. Запреев С. С., Полюдов П. А. Реализация языка Кобол на МВК Эльбрус.— Препринт/ИТМ и ВТ АН СССР.— М., 1987.— № 8.— 16 с.
43. Чеблаков Б. Г. Основные принципы реализации системы Ада-Эльбрус // Применение и реализация языка Ада: Тез. докл. 1 Всесоюзн. науч. конф.— Рига: ИК АН Латв. ССР, 1987.— С. 27—29.
44. Реализация языка Паскаль для МВК Эльбрус/Вдовкин С. В., Кубенский А. А., Лавров С. С., Сафонов В. О. // Программирование.— 1981.— № 3 — С. 62—64.
45. Вдовкин С. В., Кубенский А. А., Сафонов В. О. Реализация языка Clu // Прикладная информатика.— 1984.— Вып. 2.— С. 127—130.
46. Рейтсакас А. А. Реализация языка Лисп для МВК Эльбрус // Программирование.— 1984.— № 1.— С. 53—57.
47. Кубенский А. А., Сафонов В. О. Возможности языка АБВ и его реализация // Программирование.— 1982.— № 2.— С. 64—72.
48. Дмитриева М. В., Лаврищева В. А. Реализация языка Рефал для МВК Эльбрус. // Вестн. ЛГУ. Мат., мех., астрон.— Л.: ЛГУ, 1985.— 27 с.
49. Рябинин В. Н., Самойлов В. Ю. Диалоговый транслятор с Алгола-60 для МВК Эльбрус // Программирование.— 1982.— № 4.— С. 83—87.
50. Болдинова Н. Н., Смертин А. Н. Методы управления структурами данных и памятью в трансляторе Снобол-Эльбрус // Программирование.— 1985.— № 1.— С. 44—49.
51. Броль В. В., Гущин В. М., Яковлев В. Б. Система программирования Алгол-68-Эльбрус.— Препринт/НФ ИТМ и ВТ АН СССР.— Новосибирск, 1985.— № 14.— 36 с.
52. Дженибалаев Х. Д., Шердер Т. А., Зеленский В. Б. Транслятор языка Симула-67 в среде МВК Эльбрус.— Препринт/НФ ИТМ и ВТ АН СССР.— Новосибирск, 1985.— № 17.— 22 с.
53. Шоу А. Принципы разработки программного обеспечения.— М.: Мир, 1982.
54. Майерс Г. Архитектура современных ЭВМ.— М.: Мир, 1985.
55. Бауэр Ф. Л., Гооз Г. Информатика.— М.: Мир, 1976.
56. Уичмен Б. А. Некоторые аспекты эффективности языков системного программирования // Создание качественного

программного обеспечения.— Новосибирск: ВЦ СО АН СССР, 1978.— С. 57—69.
57. Уичмен Б. А. Производительность // Мобильность программного обеспечения.— М.: Мир, 1980.— С. 157—171.
58. Кахро М. И., Калья А. П., Тыугу Э. Х. Инструментальная система программирования ЕС ЭВМ (ПРИЗ).— М.: Финансы и статистика, 1982.
59. Бабаев И. О., Новикова Ф. А., Петрушина Т. И. Язык Декарт — входной язык системы Спора // Прикладная информатика, 1981, Вып. 1.— С. 35—73.
60. Лавров С. С. Основные понятия и конструкции языков программирования.— М.: Финансы и статистика, 1982.
61. Вейнгаарден В. и др. Пересмотренное сообщение об Алголе-68.— М.: Мир, 1980.
62. Язык программирования Ада.— М.: Финансы и статистика, 1981.
63. Клещев А. С., Темов В. Л. Язык программирования Инф и его реализация.— Л.: Наука, 1973.
64. Пратт Т. Языки программирования: разработка и реализация.— М.: Мир, 1979.— С. 60—61.
65. Лавров С. С., Капустина Е. Н., Селюн М. И. Расширяемый алгоритмический язык АБВ // Обработка символьной информации.— Вып. 3.— М.: ВЦ АН СССР, 1976.— С. 5—53.
66. Оллонгрен А. Определение языков программирования интерпретирующими автоматами.— М.: Мир, 1977.
67. Глушков В. М., Михновский. С. Д., Рабинович З. Л. ЭВМ со структурной интерпретацией языков высокого уровня // Кибернетика.— 1981.— № 4.— С. 73—81.
68. ЭВМ пятого поколения.— Материалы симп. японского общества по обработке информации: Пер. с яп.— М.: ВЦП, 1982.— Т. 3.— Гл. 1, Т. 6.— Гл. 6.
69. Шура-Бура М. Р. Система интерпретации ИС-2 // Интерпретирующие системы и элементарные функции.— М.: ВЦ АН СССР, 1965.— С. 4—25.
70. Камынин С. С., Любимский Э. З. Алгоритмический машинно-ориентированный язык АЛМО // Алгоритмы и алгоритмические языки.— Вып. 1.— М.: ВЦ АН СССР, 1967.— С. 5—58.
71. Гололобов В. И., Чеблаков Б. Г., Чинин Г. Д. Описание языка ЯРМО.— Препринт/ ВЦ СО АН СССР.— Новосибирск, 1980.— № 247, 248.— 79 с.
72. Михелев В. М., Вершубский В. Ю. АСТРА — язык для записи алгоритмов системного программирования и трансляции.— Препринт / ИПМ АН СССР.— М., 1974.— 66 с.
73. АЛЬФА — система автоматизации программирования: Сб. статей/Под ред. А. П. Ершова.— Новосибирск: ВЦ СО АН СССР, 1967.— 308 с.
74. Ершов А. П. Проектные характеристики многоязыковой системы программирования // Кибернетика, 1975.— № 4.— С. 11—27.
75. Покровский С. Б. О техническом решении проекта БЕТА // Языки и системы программирования.— Новосибирск: ВЦ СО АН СССР, 1979.— С. 71—84.

76. Гонца М. Г. Многоязыковые транслирующие системы. Синтаксис. Семантика. Проектирование.— Кишинев.: Штиинца, 1978.
77. Программное обеспечение МВК Эльбрус. (Библиографический указатель)/Евстигнеев В. А., Радченко Г. Л., Сытникова Н. И., Черемных Н. А.— Новосибирск.: НФ ИТМ и ВТ, 1987.— 29 с.
78. Головкин Б. А. Многопроцессорные вычислительные комплексы Эльбрус. (Обзор) // Программирование, 1986.— № 4, 5.— С. 76—87.
79. Computer Languages/Lanigan M. Syst. Technology, 35, 1982.
80. Computer architecture: some old ideas that have not quite made it yet /Dening P.— CACM, 24, 1981.
81. Trends in the design and implementation of programming languages/Wulf W. Computer, 1, 1980.
82. A hierarhy of higher order languages for system programming/ Lyle Don M. — В кн.: Proc. of SIGPLAN symp. on languages for systems implementation.— SIGPLAN Notices 1971, 6, 9.
83. Evaluation of the SPUR lisp architecture/Taylor G.S. et. al.— В кн.: Proc. 13th Int. Symp. on Computer architecture, 1986.
84. Architecture of SOAR: Smalltalk on a RISC/Ungar D., Blau R., Foley P., Samples A. D., Patterson D.— В кн.: Proc. 11th annual Int. symp. Computer architecture, 1984.
85. EULER: A generation of Algol and its formal definition/Wirth N., Weber H., CACM, 1966, 9, 1.
86. Abstraction mechanism in Clu/Liscov B. et. al. CACM, 20, 8.
87. Modula a language for modular multiprogramming/Wirth N. Software practice and experience, 1977, 7, 1.
88. Report on the programming language Euclid/Lampson B. W., Horning J. J. et. al. SIGPLAN Notices, 1977, 12, 2.
89. Tartan-language design for the Ironmen requirements/Shaw M., Wulf W., et. al. SIGPLAN Notices, 1978, 13, 9.
90. Programming in POP-2/Burstall R. M. et. al.— Edinburg university press, 1971.
91. Varieties of programming language/Stratchey S.— В кн.: International Computer State of the Art Report, report 7, higher level language, 1972.
92. BLISS — a language for system programming/Wulf W. et. al. CACM, 1971, 14, 12.
93. Introduction to Oregano/Berry D. M.— В кн.: ACM SIGPLAN Proc. of a symp. on data structures in programming languages.— Univ. of Florida, 1971.
94. The treatment of data types in EL/1/Wegbreit B. CACM, 17, 5, 1974.
95. The description of the Buddy Algorithm in some MOL's/ — В кн.: Machine oriented languages.— N. H., 1974.
96. The two-level approach to data definition and space management in the Lis system/Ichbiah J. P. et. al.— В кн. Proc. of ACM SIGPLAN — SIGOPS Interface meeting.— SIGPLAN Notices, 1973, 8, 9.
97. The programming language PS440 as a tool for implementating a time-sharing system/Sapper C. R. SIGPLAN Notices, 1971, 6, 9.
98. A study of Mary's data types in a system programming application/Conradi R., Holager P.— В кн.: Proc. of the IFIP Work

conf. on machine oriented higher level languaguages.— N. H., 1974.
99. Pascal user manual and report/Wirth N.— Berlin: Springer Verlag, 1975.
100. The C programming language/Kernighan B. W., Ritche D. M.— Prentice Hall, 1980.
101. Reguirements for Ada programming environments. «STONEMAN». Feb. 1980.
102. UK Ada support study. Final technical report.— Losytel, Bern, 1982.
103. CADES — software engineering in practice/McGuffin R.— В кн. IEEE Proc. on the 4 Intern. Conf. on Software engineering, Munich, Sept., 1979.
104. Can programming be liberated from the von Newman style?/ Backus Y. CACM, 21, 8, 1978.
105. Programming in Prolog./Clocksin W. F., Nelish C. S.— Springer Verlag, N. Y., 1981.
106. GEDANKEN — a simple typeless language based on principle of completeness and reference concept/Reynolds Y. C. CACM, 1970, 13, 5.
107. Recursive functions of symbolic expressions and their computation by machine/McCarthy. CACM, 1960, 3.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Научное издание

ПЕНТКОВСКИЙ Владимир Мстиславович

ЯЗЫК ПРОГРАММИРОВАНИЯ Эль-76

Принципы построения языка
и руководство к пользованию

Заведующий редакцией *А. С. Косов*
Редактор *Г. А. Слепнева*
Художественный редактор *Т. Н. Кольченко*
Технические редакторы *И. Ш. Аксельрод, С. Я. Шкляр*
Корректоры *И. Я. Кришталь, М. Л. Медведская*

ИБ № 35548

Сдано в набор 16.12.88. Подписано к печати 04.08.89. Т-11003. Формат 84×108/32. Бумага тип. № 2. Гарнитура литературная. Печать высокая. Усл. печ. л. 19,32. Усл. кр.-отт. 19,53. Уч.-изд. л. 25,9. Тираж 31 500 экз. Заказ № 9—427. Цена 1 р. 60 к.

Ордена Трудового Красного Знамени
издательство «Наука»
Главная редакция физико-математической литературы
117071 Москва В-71, Ленинский проспект, 15

Ордена Октябрьской Революции
и ордена Трудового Красного Знамени
МПО «Первая Образцовая типография»
Госкомиздата СССР
113054 Москва, Валовая, 28

Отпечатано на полиграфкомбинате ЦК ЛКСМ Украины
«Молодь» ордена Трудового Красного Знамени
издательско-полиграфического объединения
ЦК ВЛКСМ «Молодая гвардия». 252119 Киев-119,
ул. Пархоменко, 38

SUMMARY

The authour is a Soviet specialist on high level systems programming languages development, programming support environment development and their architectural support. He is the creator of «El-76» language, which is the basic programming tool of the USSR-made «Elbrus» multiprocessor system. He received Ph. D. degree and became State Prise Winner, all two in computer science. He is an Assistant Professor of Computer Department at Moscow Physical-Technical Institute.

The book describes the language aspects of the «Elbrus» multiprocessor system development, the concepts of the construction of the «El-76» high level programming language, which is used as the basic programming language of the system and also the principles of developing architectural support for an up-to-date programming support environment (PSE). The first chapter of the book analyses the basic mechanisms of high level languages (HLL) and introduces the basic constructions of hypothetic HLL, the novel characteristics of which are the combination of dynamic type control with the mechanism of abstract data types. The chapter also defines the requirements for high level languages aimed at systems programming. In order to simplify PSE development, architectural support is introduced. It is conceptually ephasized by hypothetical language virtual machine description. Virtual machine supports not only the execution of the language construction, but also PSE, which communicates with the user, using source program terms. It is especially important during debugging and when it is necessary to output a post-mortem dump, error messages and current state of programme processing. Further, the chapter analyses the interconnection between computer architecture on the one side and development of HLL and PSE on the other, discusses the limitations of well-known HLL, compared with hypothetical language mechanisms and the reasons for these limitations, analyses the reasons for the complications which arise during development of a user-friendly PSE, and the reasons for HLL non-reliability and their implementation. It is noted that in all cases the main reason is the semantic gap between HLL mechanisms and traditional computer architecture. As a result of the analysis of architecture development tendencies, one of the perspective directions is connected with the elimination of the semantic gap. The chapter demonstrates the necessity for joint development of a new language and new architecture, oriented at the main HLL mechanisms, but not at the existing HLL. The second and third chapters include a description of the El-76 language. El-76 is a concrete language, the origin of which is hypothetical language, described in chapter I. The fourth and fifth chapters,

as well as appendicies, contain informatic data, which can be used in language utilization (object atributes, implementation dependent features, symbol coding, language syntax, etc.). The book is dedicated to programmers, computer specialists and readers, interested in modern tendencies of computer and language development. The book has taken account of the latest development in HLL-oriented architecture, programming support environments and system programming languages.

What puts the book in a class by itself is the suggestion of the fundamentals for joint construction of computer architecture and HLL, oriented at wide range of utilization, starting from operating system programming to scientific application programming. El-76 language, described in the book has implemented this idea and has been used for more than ten years. Some million lines of various system and application programmes have been programmed with El-76, including Elbrus operating system, PSE, data base systems, scientific application packages and etc.

It is the second edition of the book. The main changes of the second edition compared with the first edition is inclusion of new El-76 features, concerning dynamic abstract data types control and its conceptual model. Besides that the description of El-76 has been brought in conformity with the current state of Elbrus operating system facilities.

Against the academician of the USSR Academy of Sciences A. P. Ershov opinion, «Elbrus architecture gives outstanding extension for powerful general purpose computer idea... Due to modularity and multiple processors, the two Elbrus models cover two levels on the performance scale. Powerful runtime checking facility and other structural decisions allow to use only one safety standard, ensuring reliability for wide application range. The basic operating system without any options and additions provides various processing modes and their combination: batch multiprogramming, time sharing, multiprocessing and network connection. General purpose filing system is used for user program development as well as for operating system programming... ...All the above and many other Elbrus features not mentioned here but no less interesting have emboded in basic programming language of the system. It is the language which exposes Elbrus architecture pecularities».